コンピュータ・サイエンス研究書シリーズ ⑩

コンパイラ作成の技法

コーネル大学教授 David Gries 著
九州大学情報工学科教授・工博 牛島和夫訳

日本コンピュータ協会

コンピュータ・サイエンス研究書シリーズ ⑩

コンパイラ作成の技法

コーネル大学教授 David Gries 著
九州大学情報工学科教授・工博 牛島和夫訳

日本コンピュータ協会

COMPILER CONSTRUCTION FOR DIGITAL COMPUTERS

David Gries
Cornell University

訳者序文

電子計算機が誕生してすでに30年以上を経過した．そして最初のFORTRANコンパイラがBackusらの手によってこの世に出て20年になる．ALGOLの最初の予備報告書が出されたのも1958年であった．機械語かあるいはそれに近い低水準の言語で機械の性能を十分に発揮させた（と信じて）プログラムを書いていた人達に，高水準言語を受入れてもらうため，人間が書いたと同程度に能率のよい目的プログラムを，このFORTRANコンパイラが生成できるということを，Backusらは強調している．現在では，高水準のプログラム用言語（やコンパイラ）なしで計算機システムの使用は考えられない．機械の性能ばかりでなく，プログラムの作成過程や維持修正まで含めた人間の能率が重視されるのである．

コンパイラの歴史20年の中で1960年代は，未熟であり，手探りであったコンパイラ作成の技術が，理論化され，体系化されて，コンパイラが行うこと，それを実現する方法が明確にされた10年であった．COBOLやPL/Iもその間に生れて実用化された．わが国でもさまざまなコンパイラが試みられ実現された．1960年に大学を卒えた原著者は，コンパイラの理論と実際の成長に，その只中で関わってきた一人である．コンパイラ作成計画に加わり，Stanford大学やCornell大学などでコンパイラ構成論を講じた．その講義ノートがもとになって本書が出版されたのは1971年のことである．形式言語理論入門からコンパイラ作成技法の実際までを包括した最初の教科書といってよい．その後しばらくこのような教科書が現れなかったこともあって，計算機科学に関わる多くの大学で教科書や推薦書として広く用いられてきた．現在では増刷回数も10刷を越えている．本書を引用している研究論文も多数ある．

最初のFORTRANコンパイラの完成には，18人・年で実質2年半を要したという．20年たった現在，半年あるいは1年の講義と並行して行われる演習の中で，学生の手によって，コンパイラのひな形が容易に作られるようになった．コンパイラ作成に便利なツールも備えられるようになったこともある．作

成法の原理が理解されたとしても，このようなひな形を，プログラミングツールとして，多くの人々の使用に耐えるようにするにはまだ成すべきことがたくさんあることを弁えておかねばならない．それには，本書の15章や21章，そして下記の文献などを手掛りにして，さらに学んでいただきたいと思う：

Bauer, F. L. and Eickel, J. (Ed.) *Compiler Construction——an Advanced Course*, Springer-Verlag, New York, 1974. 特に第5章 Engineering A Compiler を奨めたい．第2版が1976年に出ている．Aho, A. V. and Ullman J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977. 巻末に21ページに亘る文献表があり，最近の文献を確かめるのに便利である．

原著者まえがきに述べられているように，原著の本文は，IBM 360/65 に接続された計算機用の印刷装置への出力をそのまま製版して印刷したものである．使用できる活字は数が限られており形もひととおりである．印刷装置にない特殊記号は，2～3文字を連結して表すなどしているため読易さに多少欠ける点がある．このような記号は，訳書では伝統的なものに合せるよう置換えた．そのほか，計算機の出力のため l と1の区別など不鮮明な部分があり，少なくともこれらの点で訳書の方が読易くなったのではないかと思う．

プログラムや図は，誤りの混入をさけるために原著のものをできるかぎり使用したかった．しかしこれらの大部分も計算機の出力であり，そのまま用いると印刷リボンのかすれが目立ったりする．そこで思い切って，プログラムは新たにタイプし直し，図はトレースし直した．校正には殊のほか気を使うことになったが，この過程でも原著の小さな誤りをいくつか見付けて訂正することができた．訳書の出版はもっと早くを予定していたにも拘らず翻訳原稿の完成から丸2年も要してしまった理由の一つがここにある．期間が長くなったために仕事に関わっていた三つの印刷工場の一つが経済情勢の影響を受けて仕事が不能になるなどの事態が生じ，さらに出版を遅らせることになったのは残念である．

原著の内容をわれわれの言葉で正しく伝えるべくできるかぎり努めたつもりであるが，いたらぬ点もいろいろあるかと思う．読者の方々の御批判をお願いする．翻訳に当ってつぎの方々に特にお世話になった．形式言語理論と自動構文解析に関連する各章は，原稿の段階で九州大学理学部の有川節夫助教授に目を通していただき懇切な御注意を頂いた．また工学部の荒木啓二郎君には，原

著と翻訳を逐一対照するという面倒な作業をお願いし不備を指摘してもらった．ここに記して感謝する．最後に出版に際して終始御助力をいただいた村越勝弘氏を始め科学技術出版社の方々に謝意を表したい．

1978年5月

牛島和夫

まえがき

コンパイラやインタプリタは，どんな計算機システムにとっても不可欠なものであり，これがなければわれわれは皆アセンブリ言語や機械語でプログラムを書かねばならない！ このように，コンパイラの作成は計算機科学の中で重要かつ実際的な分野である．本書は一貫して，コンパイラ作成に用いられる代表的な技法を提示し，初学者に対してはこの分野に入ることを容易にし，経験者に対しては，文献の参照を容易にすることを目的としている．

本書はいわゆる構文向きコンパイル法を指向している．実際，本書の三分の一以上の分量が形式言語理論と自動構文解析の話題にあてられている．コンパイラ作成に携わっている人はだれでもこの問題に関する基礎的な知識を持つべきであると著者は強く感じている．だからといって，コンパイラは全て自動構文解析法を用いて書かねばならないと云っているのではない．このような方法が適さないようなプログラム用言語もたくさんある．しかし，形式言語理論に関する基礎的な知識を持っていれば，コンパイラ作成者は，自分の作成しているコンパイラの中で起っていることをより深く洞察することができるようになり，この知識がコンパイラの設計とそのプログラム作成をより組織的かつ能率的に行う助けになるにちがいない．

しかしながら，構文解析はコンパイラ作成のごく一部にすぎない．したがって本書にはコンパイラ作成の重要な話題——記号表の構成法，誤り回復法，コード生成法，コードの最適化法など——をすべて包含した．定数の変換とか，積上げ式コンパイラのような話題はいくつか省略した．本を適当な大きさにとどめるためである．

本書は二様の要求に合致するように計画してある．すなわち，一つは，コンパイラ作成に関心を持ったり，携わったりしている職業的プログラマに対する自習書，参考書として用いることができるよう，もう一つは，コンパイラ作成に関する大学の１学期の講義用教科書として使用できるようにしている．事実，本書は，ACM のカリキュラム委員会が Communications of ACM の

1968年3月号で推奨しているI5 課程（コンパイラ構成論）で列挙している話題全て（さらにそれ以上）を網羅している．

読者には，高水準言語（例えば，FORTRAN, ALGOL, PL/I など）とアセンブリ言語によるプログラム作成経験が少くとも1年はあり，ALGOL で書かれたプログラムを読んで理解することができることを前提としている．一部では，初等的なブール行列の理論の知識を仮定している（手短かな説明はつけた）．読者には，集合の意味，2個の集合の合併，などに関する知識があるものとしている．さらに，大学1，2年程度の数学の素養を備えていることが望ましい．†

高水準言語によるプログラム経験が必要なことは明らかであろう．なにしろ本書の目的はこのような言語で書かれたプログラムを翻訳しようというのである．計算機が実際どう動くかを完全に理解するためには，アセンブリ言語の経験はより重要である．しかし実際には，特定のアセンブリ言語にはほとんど言及しないつもりである．本書には，数ケ所，IBM 360 のアセンブリ言語のプログラムが挙げてあるが，読まないで飛ばしても理解の妨げにはならない．

コンパイラそれ自身も，適当な言語で書かれたプログラムである．したがって，コンパイラの各部の例は，適当なプログラム用言語で書いておかなければならないので，読み易さの点からいって ALGOL 型言語を選んだ．例に挙げたプログラムは，普通は非常に短いものばかりでそれをたどるのは大変容易である．プログラムがあまりに詳細に渉ると当面の問題の見通しがかえって悪くなるので，そこでは ALGOL の代りに，自由に普通の言葉を用いた．これらのプログラム例は，紙上で十分検査しているが，必ずしも計算機の上でデバッグしたものでないことを警告しておこう．

本書で用いている ALGOL 型言語（擬 ALGOL と称する）の簡単な説明を巻末の補遺に示した．この説明は短いもので，ALGOL の知識を大いに期待している．もし読者が ALGOL や，言語の構文記述法に不慣れならば，第2章を学んでから補遺を読むことを奨めたい．

本書には，通常の1学期の課程で学ぶ以上の分量が入っている．最小構成としてつぎのような組合せを推したい．

第1章．　はじめに

---

† **訳注**　ACM（Association for Computing Machinery 米国計算機科学協会）のカリキュラム68 では I5 コンパイラ構成論の前に学ぶべき科目として，B1 計算法入門，B2 計算機とプログラミング，B3 離散構造入門，I1 データ構造論，I2 プログラミング言語を推している．

第2章.　文法と言語（7節を除く）
第3章.　走査部（4節, 5節, 6節を除く）
第4章.　下向き構文解析法（とくに3節は必修）
第5章.　単純順位文法
第8章.　実行時における記憶域の構成（6節，9節を除く）
第9章.　記号表の作成
第10章.　記号表中のデータ（2節を除く）
第11章.　原始プログラムの内部表現（4節，5節を除く）
第12章.　意味付けルーチン入門
第13章.　ALGOL 型文法要素に対する意味付けルーチン（6節を除く）
第14章.　実行時における変数の記憶場所の割付（3節を除く）
第16章.　インタプリタ
第21章.　コンパイラ作成者への指針

読者は，著者が，構文解析に対して単純順位技法を強調していることに気が付かれるであろう．これはこの方法が最善のものだという理由ではなく（おそらくは，最悪のものかもしれない），これが最も教え易いからである．もっと時間があれば，教授者は，上向き構文解析法，演算子順位法（6.1節），高階の順位法（6.2節），推移行列法（6.4節），生成言語による方法（第7章），あるいは本書にのっていない方法，のうち好みのものを加えて教えることをお奨めしたい．

教材の順序も変更して構わない．実際，講義に用いる場合，構文法の理論を勉強する場合には，これをいくつかの実用的な素材に分割するのが良い．第8章の実行時の記憶域の管理の章は，独立しているが，第9，10，11，16章は記号表，原始プログラムの内部形式，インタプリタに関するもので，いずれにしろ，この順に学んでもらいたい．

第21章は特別に注意を払う必要がある．この章は，コンパイラ作成者が弁えておくべき各種の実例や意見を集大成している．これらのものは，ほかのどの章にも属さないか，ほかの章の中にうずもれるにはあまりにも重要なことがらである．読者は，この章に時に応じて目を通し，その時の興味ある節を読んだらよい．

コンパイラ構成論の講義は，実習を伴ったものでなければならない．学生は，2，3人のグループで，適当な簡単な言語に対するコンパイラなりインタ

プリタなりを書きデバッグする必要がある．そのようにして初めて，コンパイラの中で行われていることがらを本当に理解するようになる．インタプリタが最も良いだろう．それは，面倒な機械語の詳細に関わる必要がないからである．大切なのは考え方であって細部ではない．この方針に従えば，コンパイラ作成の全体は高水準言語でプログラムすべきである．著者の経験では，PL/IやALGOL型言語の方がFORTRANより適している．FORTRANで書いたコンパイラは，大きくなり，さらに読み難くなりがちである．もし可能ならばコンパイラ作成用言語を使用したらよい．

何か適当な変化を付け創造力を発揮させるためには，整数型変数，代入文，式，名札と分岐，条件文，そして簡単な入出力文を含んだ基礎的で簡単な言語から始めて，グループごとに，この言語に，二，三の要素を付け加えて拡張させる．例えば，配列，レコード（構造体)，各種のデータ型，ブロック構造，手続き，マクロ，繰返し文などである．

このコンパイラは，講義の進行と並行して，書きながら検査してゆくことができる．最初に走査部，つぎに構文解析部，そして記号表処理ルーチンである．インタプリタは，これを扱った章を終れば直ちに設計し作成することができる．このように，コンパイラ作成の演習は，学期を通して平均して行うべきもので，学期末にひとまとめに仕上げるようなものではない．

参考文献は大部分，各章の最後の節に与えてある．もつとも，別の個所にあるものもいくつかある．文献の著者の名前が自動的に，文献表中の文献を参照している．この文献表は，著者名のＡＢＣ順で，著者の内部では年代順に整理してある．一人の著者のものが２篇以上の場合は，〈名前〉（〈年代〉）の形式で表している．ここで〈年代〉とは，出版の年である．例．Gries (68)．同じ年に２篇以上の出版物がある場合は，最初のものを（〈年代〉ａ），つぎを（〈年代〉ｂ）...のように引用する．したがって，Floyd (64ｂ）は，プログラム用言語の構文に関するFloydの論文を引用している．

表題やいくつかの図版を除いて，本書はIBM 360/65で，Gerald M. Berns (69) の作ったFORMATと呼ばれるプログラムを用いて作成された．さらにJohn Ehrmanにも負うところが大きい．彼は，このプログラムの重要な変更と追加をいくつか行ってくれた．FORMATを使ったので，本書のもとの材料を編集しそれを学生諸君に配布するのが容易になった．しかしながら，これを使ったために，普通用いている記法をやむなく変更せざるを得ない場合があった．主な変更は，つぎの２点である．本書の印刷に用いた装置の鎖には，上付

き添字も下付き添字もないので（ただし ^{0} から ^{9} はある），べき乗は演算子 ! を用いて表わした.† すなわち $c\,!\,b$ は c の b 乗である．添字がないので，n 個の記号の系列は，$S[1]$，$S[2]$，…，$S[n]$ と表さざるを得なかった.† 前後関係から意味が明らかなところでは，単に $S1$，$S2$，Sn と書いている．

本書の素材は，Stanford 大学と Cornell 大学のコンパイラ構成論の講義ノートから始っている．このノートの改訂したものを Ann Arbor の Michigan 大学夏期講習や Cornell 大学，1970 年イスラエルで行われた上級プログラミングシステムの国際セミナーで用いる機会を得た．これらのコースに出席した聴講生諸君のこのノートに対するさまざまな貴重な意見に感謝したい．また多くの方々から，助言をいただいた．なかでも，Richard Conway，Jerry Feldman, John Reynolds, Bob Rosin, Alan Shaw にはお世話になった．また Steve Brown には，私の原稿を注意深く，すみずみまで読んで誤りを見付け，貴重な注意と比評をしていただいたことを心から感謝する．

最後に，私の妻にも感謝の意を表したい．妻は，私が本書を書いている間驚くべき寛容と理解を示してくれた．

† **訳注** 訳書は計算機の出力ではないので，普通に用いている記法に従った．ただし一部に原著の図をそのまま写している箇所で計算機の出力と同じ形式のものがある．

目　次

第1章

はじめに

1.1 コンパイラ　アセンブラ　インタプリタ

トランスレータ (translator) とは，**原始プログラム** (source program) をそれに等価な**目的プログラム** (object program) に翻訳するプログラムである．原始プログラムは，**原始言語** (source language) で書かれており，目的プログラムは，適当な**目的言語** (object language) で書かれている．トランスレータそのものが実行される時を，**翻訳時** (translation time) という．

原始言語が，FORTRAN，ALGOL，COBOL などの高水準言語であって，目的言語が特定の計算機のアセンブリ言語や機械語である場合に，トランスレータを**コンパイラ** (compiler) と呼ぶ．機械語は，時には，**コード** (code) と呼ばれる．したがって，目的プログラムのことを**目的コード**と呼ぶこともある．原始プログラムを目的プログラムに翻訳する時を**コンパイル時** (compile time) と呼び，目的プログラムが実際に実行される時を**実行時** (run time) と称する．

アセンブラ (assembler) は，計算機ごとに異なったアセンブリ言語で書かれた原始プログラムをその機械語に翻訳するものである．アセンブリ言語は機械語に極めて似ており，事実，アセンブリ言語の大抵の文は機械語の命令を記号で表現したものにすぎない．そしてアセンブリ語の文は，固定した書式になっており，そのため，その分析は容易である．またブロック構造のように入れ子を作る文は存在しないのが普通である．

原始言語に対する**インタプリタ** (interpreter) は，原始言語で書かれた原始プログラムを入力してそれを直ちに実行するプログラムである．コンパイラとインタプリタのちがいは，インタプリタは，目的プログラムを生成せず，原

始プログラムをそのまま実行してしまうところにある．

純粋な意味でのインタプリタは，実行すべき原始プログラムの文に出会うたびにその文をどのように実行すべきかの解釈を行う．このようなインタプリタは非常に能率が悪いし，そう頻繁に使われることはない．普通に行われているのは，インタプリタを二つの相に分割する方法である．第一の相では，コンパイラが行っていると同様に，原始プログラムを解析し，それを内部形式に変換する．第二の相では，内部形式になった原始プログラムを解読しながら実行してゆく．この内部形式は，それを実行するための“解読”に要する時間が最小になるように設計することが肝要である．

先に述べたようにコンパイラはそれ自身，原始プログラムを入力とし，それに等価な目的プログラムを出力とするプログラムであって，何らかの言語で書かれている．歴史的に見れば，コンパイラは，個々の計算機のアセンブリ言語で書かれていた．当時は，使用できる唯一の言語は大抵の場合アセンブリ言語であった．しかし，次第にコンパイラを高水準言語で書くようになってきた。こうすればプログラムを書く時間を減らし，書き上ったコンパイラを読易くすることができる．今日では，コンパイラを書くことを表明して設計された言語がたくさんある．これらのいわゆる“コンパイラ-コンパイラ”は，“トランスレータ作成システム”(**t**ranslator **w**riting **s**ystem；略して **TWS**)の一種である．これについては，第20章で簡単にふれることにする．

本書は，コンパイラ作成技法の入門書である．インタプリタを作る際の問題点には触れてはいるものの比較的わずかである．それは，コンパイラを作る際の技法とインタプリタを作る際に使われる技法とは大部分共通だからである．アセンブラについては議論しないが，コンパイラの作成技法を理解したものならば，アセンブラが何を実行し，いかに働くかを理解するのに困難はないはずである．

本書では，完全なコンパイラはどこにも提示していない．何故ならば，本書は，著者が，特別なコンパイラをどう書いたかを示すのではなくて，読者が自分で自分のコンパイラを書く方法を学ぶことを主眼としているからである．本書には，コンパイラ作成に使われる多くの技法に関連する例題や，議論が示してある（もちろん，これが全てというわけでなく，また大部分というわけでもない)．例題は，ALGOL 型言語で記述している．この言語については，巻末の補遺で簡単に説明している．もし本書を教科書として用いようという場合には，読者自身で，自分のコンパイラなりインタプリタを，ALGOL なり FOR

TRAN なり，PL/I なり，あるいはほかの高水準言語なりで書いてみることを希望する．これがコンパイラ作成技法を学ぶ最善の方法である．

1.2 コンパイル過程の概観

コンパイラはまず，原始プログラムの**解析**を行い，つぎに目的プログラムの**合成**を行わなければならない．すなわち，まず原始プログラムを，基本的な要素に分解する，次にそれを等価な目的プログラムに作り上げる．これを行うために，コンパイラは，解析相の間に，いくつかの表を作成する．これらの表は，解析，合成の二つの相の中で使用される．図1.1は，コンパイル過程の全体をやや詳しく示したものである．ここで点線は情報の流れであり，実線はプログラムの流れである．以下にコンパイラのそれぞれの部分を簡単に説明しよう．

各種の表

原始プログラムの解析が進むにつれて，宣言や手続きの頭部，あるいは，for-ループなどから情報が得られ，これを後で使用するためにとっておかなければならない．これらの情報は，その場で局所的に調べられ，後にコンパイラの全ての部分から使用できるように集積される．例えば，ある**名前** (identifier)† を使うたびに，その名前がすでに宣言されていたか，よそでどう使われていたかを知る必要がある．どんな情報をとっておかなければならないかは，厳密には，原始言語や目的言語，あるいはコンパイラの複雑さなどによって変わってくる．しかし**記号表** (symbol table, **名前一覧表** identifier list, **名前表** name table などの呼び方もある）は，ともかく，全てのコンパイラが用いるものである．これは，原始プログラムで使われている全ての名前を，その属性とともに記録しておく表である．属性というのは，その名前の，型であったり，目的プログラムにおける番地であったり，そのほかコードを生成するのに必要な各種情報である．

ほかにどんな情報を集めなければならないのであろうか．原始プログラムで使われている定数の表は，大抵の場合必要となる．この表は，定数それ自体とそれをしまうべき目的プログラム中の番地を含んでいるだろう．for-ループのテーブルも必要となるだろう．これには，入れ子構造とループの制御変数に関

---

† **訳註** identifier に対する訳語として，ALGOL，COBOL の JIS 規格ではそれぞれ名前，一意名が用いられている．FORTRAN では**英字名** (symbolic name) と名付けている．

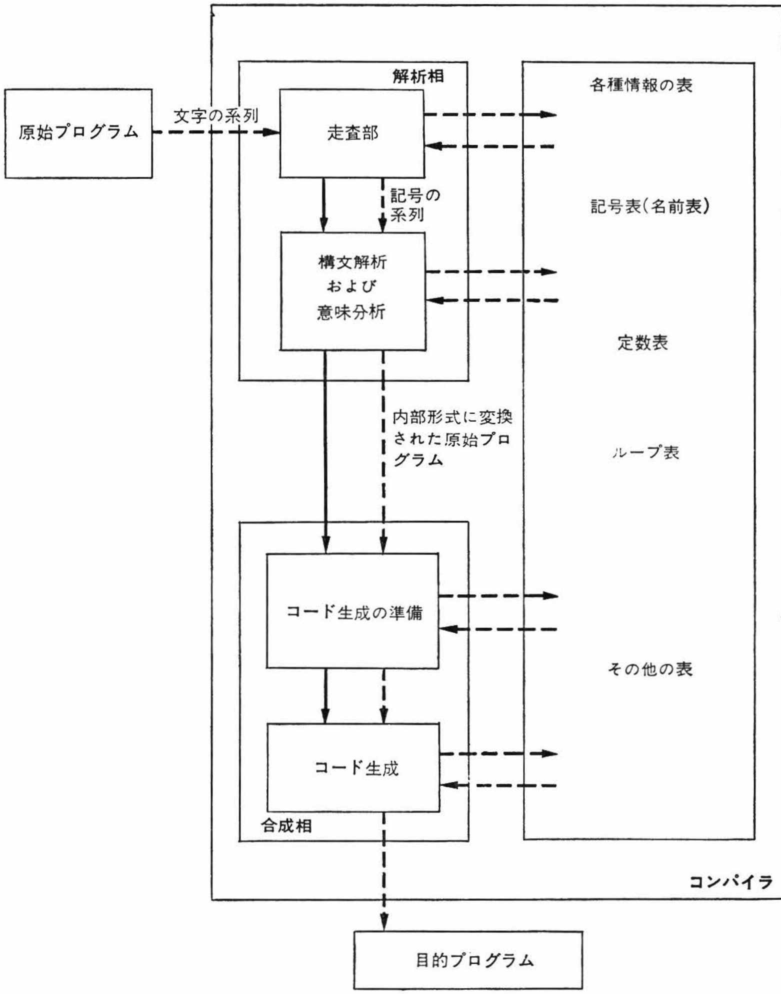


図1.1　コンパイラの論理的な関係

する情報を記録する．また FORTRAN などの EQUIVALENCE 文に関する情報や各手続きの目的プログラムの大きさなどに関する表も必要であろう．コンパイラの設計にあたって，原始プログラムの各文と目的コードとの対応やコンパイラの合成部の内容がかなり詳細にならなければ，集めておかなければ

ならない情報の内容や形式は決定できない．さらにこれらは目的コードの最適化の程度にも大いに依存している．

走 査 部

走査部 (scanner) あるいは**字句解析部** (lexical analyzer) は，コンパイラの中で最も単純な部分である．これは原始プログラムの文字列を左から右に走査して，プログラムの意味のある**記号** (symbol)——すなわち，整数，名前，予約語，＊＊や／／のような2文字記号など——を組立てるものである（時には，記号のかわりに**しるし** (token) とか**原子記号** (atom) のような呼び方をすることもある）．つぎにこれらの記号を実際の**解析部** (analyzer) にわたす．注釈の部分は削除してもよい．走査部は，名前を記号表にのせたり，そのほか，実際に原始プログラムを解析しなくてもできる簡単な仕事をすることもある．文字列の挿入だけを司さどるマクロの処理の大部分も行うことができる（マクロの処理については，第19章参照）．

記号は，通常内部形式に変換して走査部が直接解析部にわたす．区切り記号（予約語，演算子あるいは句読点など）は全て整数で表現されることになろう．名前や定数は，番号の対として表現できる．その対の一方は，区切り記号を示す整数とは異なった整数で“名前”または“定数”であることを明示するものである．もう一方は，名前や定数用の表の該当する番地や指標を示すものである．このようにしておけば，コンパイラがのちに，可変長の文字列の代りに，固定長の記号を扱うことができて大変能率がよくなるのである．

構文解析および意味分析

これら二つの解析部は，つぎのような実質的にこみいった仕事を受持つ，すなわち，原始プログラムを，それを構成している部分要素に分解し，内部形式を作成し，記号表などの表に情報を書込む．構文や意味の検査も完全に行う．

標準的な解析部の働きは，プログラムの構文によって制御される．事実，構文と意味とは，できるかぎり分離するような傾向になっている．構文解析部（パーサ，parser）が原始プログラムの構成要素を認識すると，解析部は，いわゆる**意味付け手続き** (semantic procedure) あるいは，**意味付けルーチン** (semantic routine) を呼出す．これは，この構成要素を受取って意味上の正しさを検査し，それに関する必要な情報を記号表に書込んだり，そのプログラ

ムの内部形式を作成したりする．例えば，簡単な宣言を認識すると意味付けルーチンは，そこで宣言されている名前がすでに宣言されていないかどうかを確かめ，宣言された属性とともに記号表に書き加える

〈変数〉　:＝　〈式〉

という形式の代入文を認識すると，意味付けルーチンは，〈変数〉と〈式〉の型の一致を検査し，代入文をプログラムの内部形式に変換する．

原始プログラムの内部形式

原始プログラムの内部表現は，後でそれをどのように使うかによって大きく変わってくる．この表現は，原始プログラムの構文を表す木表現でもよい．**ポーランド記法**と呼ばれるものでもよい．また（演算子，被演算子，被演算子，結果）というような 4-組の形式もある． これは， 実行される順序に並ぶことになる．例えば，代入文“A=B+C*D”は

```
*,C,D,T1
+,B,T1,T2
=,T2,,A
```

となるだろう．ここで，T1，T2 はコンパイラが作りだした中間結果用の変数である．上例で被演算子は，記号としての名前そのものではなく，記号表中でその被演算子を記述している場所へのポインタになるだろう．

コード生成の準備

コードを生成する前に，内部形式プログラムを何らかの方法で処理したり変更したりするのが一般的である，すなわち，実行時における変数の記憶場所の割付を行わねばならない．FORTRAN では，COMMON 文や EQUIVALENCE 文を処理しなければならない．この過程で重要なことは，目的プログラムの実行時間を減らすためのプログラムの**最適化** (optimization) である．

コード生成

これは，原始プログラムの内部形式をアセンブリ言語，または機械語に，実際に変換する部分である．おそらく，ここが最もこみいった細かい部分ではあるが，理解は最も容易な部分である．上で述べたような 4-組の内部形式を

仮定すると，4-組ごとにコードを生成してゆけばよい．上の3個の4-組をIBM 360でコード生成してみると，アセンブリ言語ではつぎのようになる：

```
L  5, C    Cの値をレジスタ5に
M  4, D    Dを乗じ結果はレジスタ 4~5 に
A  5, B    上の結果にBの値を加え
ST 5, A    その結果をAにしまう
```

インタプリタの場合には，コンパイラのこの部分が，内部形式を直接解釈して実行する部分と置換わる．この場合，内部形式も，コンパイラのそれとそんなに異なったものにはなっていないであろう．

図1.1は，コンパイラの時間的な関係というよりは，論理的な関係を示している．論理的に前後の関係にある4連の処理——走査，解析，コード生成の準備，コード生成——は，図1.1のような順序で行われてもよいし，並列に（といっても，交互に）行われてもよい．どれをとるかの判定基準の一つは，使用できる主記憶の大きさである．コンパイルをいくつかのパス（原始プログラムを主記憶にロードする回数）にわけることが有利なこともしばしばあるし，必要に迫られてそうすることもある．この場合，原始プログラム以外の情報は，入出力時間を節約するために主記憶に止めおくのが望ましい．もう一つの判定基準は，コンパイルの速さ，あるいは，目的プログラムの速さ，あるいは目的プログラムにおけるデバッグの機能など，それぞれどの程度にするかといった，コンパイラ作成計画の目標である．このほかの要因として，コンパイラ作成計画へ参加する人数がある．人数が多くなると，パスの数は多くなる傾向があるが，そうしておくと，それぞれの部分に対して各人が責任を明確にできるからである．

図1.1の全ての部分が必要なわけでもない．1-パス・コンパイラでは，原始プログラムの内部形式は，必要でないし，コード生成部とその準備の部分は，意味分析部の意味付けルーチンの中に一体となって入ってしまう．図1.2に典型的な1-パスコンパイラの骨組を示した．構文解析部は，新しい記号がつぎに必要になると走査部を呼出し，これが要素の組立てを終ると意味付けルーチンを呼出す．このルーチンは，構文解析部に戻る前に組立てられた要素に対してその意味を調べ，記憶場所を割付けそしてコード生成を行う．

全ての言語が必ずしも1-パス・コンパイラで翻訳できるような構造にはなっていない．

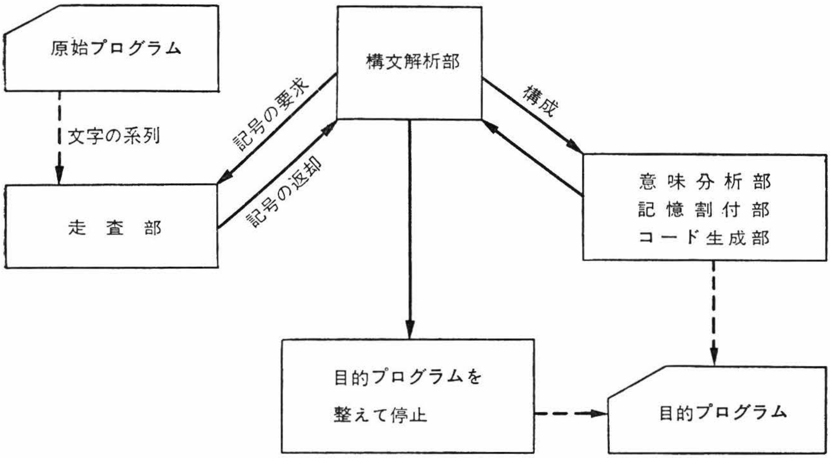


図1.2 1-パスコンパイラの例

読者は，コンパイラを作るのに，最も難かしいのはどこかとたずねるかもしれない．走査部は，ほとんどそのままで明らかなのですぐ理解できるであろう．構文解析部も，本書で扱うような簡単な形式言語に対しては，かなりよく理解できよう．実際，この部分は，大部分を自動化することができる（構文は，形式的に記述されてきたので，コンパイラ作成研究は大部分，意味の代りに構文を取扱ってきた）．最も難かしく，“泥くさい”部分が，意味分析とコード生成とその準備の部分である．これらは三つとも，相互に関連しており，かなりの程度まで，一まとめに設計してやらなければならない．その設計も目的プログラム用言語と機械が変れば，根本から変更しなければならない可能性もある．

これで，本書の第一の主題である形式言語理論とそのコンパイラ作成法への応用の話題を始めるところまで来た．次節は，これまでに述べたことの印象を強めるために現存するコンパイラの例をいくつか紹介しているが，必要がなければ，省略して先に進んでもよい．

1.3 コンパイラの構成例

ここでは，現存するコンパイラの中から4例を紹介する．すなわち計算機がちがえば，いかにコンパイラの設計に影響するかを説明する例として，ALG-OL コンパイラを二つ，さらに同じ計算機に対して作られた二つの違った FO-

RTRAN コンパイラを，コンパイラの目的とするところが違えば，設計も違ってくる例として挙げてある．

ALCOR Illinois 7090 コンパイラ [Gries (65) 参照]

これは，IBM 7090—7040 用に作った 4-パス・コンパイラである．原始言語は，ALGOL 60 から **own** を除いたものである．このコンパイラは，for-ループ中の添字の評価の最適化をうまく行っている（後に述べる FORTRAN コンパイラは，コードの最適化をさらに徹底的に行っている)．コンパイラの出力は，7090 の 2 進コード（バイナリ・デック）の形式で，それは，システム・ローダによって，主記憶にロードされ実行される．

パス	内容
1	走査部と部分的な構文解析（全ての名前に対してブロック構造を持った記号表を生成する.）プログラム中の全ての記号は，固定長の整数に置換える.
2	完全な構文および意味分析．添字付変数の最適化の準備をする．実行時の記憶割付（これは実際には合成の一部である.）
3	添字付変数の最適化とコード生成.
4	パス 3 の出力からバイナリ・デックを作成，誤りがあればエラー・メッセージを出力.

Gier ALGOL コンパイラ [Naur (63b) 参照]

このコンパイラは，わずか1024語の主記憶と 128 *K*語のドラムを持つ計算機（1 語42ビット）のために作られた．計算機の大きさにもかかわらず，コンパイラは，本質的には制限のない ALGOL を翻訳している．コンパイラは 9 個のパスにわけなければならなかったがその理由は以下でわかる．パス 1 およびパス 2 は，字句解析部であり，パス 3 は，構文解析，パス 4，5，6 は意味分析を行う（合成の一部，すなわち実行時の記憶割付もここで行われることに注意)．パス 7，8 は合成である．パス 9 は，このコンパイラ特有のもので，能率のよい目的プログラムを生成するために必須な部分である．

パス	内容
1	走査部：区切り記号（名前と定数を除く全ての記号）を内部形式に

変換する.

2 原始プログラム中の全ての名前を固定長の整数で置換える.

3 構文解析：以後の処理を容易にするために特別な区切り記号を挿入し，残りの区切り記号を変換する.

4 記号表の作成：各ブロックごとに，宣言または使用されている名をその属性とともに表に書込む.

5 各変数に実行時の記憶場所を割付ける．さらに原始プログラム中の全ての名前（整数にコード化されている）を，型，種類，ブロックの番号，実行時の番地，さらに必要があれば引数や添字の個数などからなる4バイト長の情報で置換える.

6 全ての名前とそのほかの被演算子について，型と種類を検査し，原始プログラムをポーランド記法に変換する.

7 コード生成.

8 いくつかの命令の最終的な番地割付．ドラムのトラックの配置．最終的なコード生成.

9 ドラム上のトラック配置の再調整.

/360 WATFOR [Cress ほか参照]

WATFOR は，IBM 360 に作られたバッチ処理用の FORTRAN IV コンパイラ兼サブモニタである．この目的は，大きさが平均以下で，実行時間が比較的短かい FORTRAN プログラムをすばやく処理してターン・アラウンド時間を短くしようというものである．このようなプログラムは，教育実習用でよく見かける．上の目的を達するために，全システムが主記憶の中にあって，入出力は，原始プログラムと，そのプログラムの結果の出力だけである．システムは最低 128 Kバイト（32K語）必要とする.

このコンパイラは，目的コードの最適化はしないし，比較的能率の悪い目的コードを機械語で生成し，それをシステムが直ちに実行する．このため連係編集やロードなどの時間が不要になって時間が節約される.

このコンパイラは，本質的に1-パス・コンパイラである．走査部は，1個の FORTRAN 文を，一度に内部形式に変換し，文の種類を決定し，それを処理するのに適する意味付けルーチンを呼出す．このルーチンは，その文の構文を解析し，意味を分析して，直ちに目的コードを生成してしまう．解析と合成が一時に行われるのである．FORTRAN では，実行時の記憶場所は，CO-

MMON 文と EQUIVALENCE 文を全て処理し終らなければ，完全には決められない．したがって，目的コードの番地部は，実行時の番地そのものではなくて，コンパイラの記号表の中の対応する変数や，中間結果用変数の場所へのポインタとなっている．主要なパスが終ると，第二の小さなパスが始まり，実行時の変数を全て割付け，目的プログラムの番地部の記号表へのポインタとの置換えを行う．

IBM 360 FORTRAN H コンパイラ [IBM (a) 参照]

このコンパイラの目的は可能なかぎり能率のよい目的プログラムを生成することにある．このコンパイラはこの目的を見事に遂しとげた．コードの最適化のために，コンパイラ自身の処理速度は，非常に遅い．コンパイラは，つぎのようなパスからなっている：

パス	内容
1	走査部と構文解析部：原始プログラムを（演算子-被演算子）対の内部形式に変え記号表を作成する．
2	(実質的に三つの部分パスに分れる)．
	a) COMMON 文と EQUIVALENCE 文の処理.
	b) 原始プログラムの内部形式を（演算子，被演算子，被演算子，結果）という 4-組表現に変換.
	c) 実行時の記憶場所の割付け.
3	コードの最適化.
	a) 冗長な演算を除去，可能なかぎりループの外へ演算を移動など.
	b) 実行時のレジスタの割付け.
	c) 分岐の最適化.
4	最終のコード生成.

第 2 章

文 法 と 言 語

形式言語理論を知らない読者にとって，本章は，本書の中で最も難解な章であるかもしれない．本章では，われわれが普段そのまま使っている40以上の術語，例えば“句”，“文”，“言語”，“あいまいさ”，といった術語に厳密な意味を与える．2.11節に，定義した順でこれらを一覧表にしておいた．本章を読み終えた時に，この一覧表を調べて，知識を確かめてほしい．もし完全な知識となっていないものがあれば，復習してほしい．この章を完全に理解すれば，以後の章を読むのがずっと簡単で楽しいものになるであろう．

2.1 文　法

“The big elephant ate the peanut” という文を考えてみよう．英語の知識があれば，これは英語の文であり，これを図 2.1 のように図式化することができる．

図 2.1 のように文を図式化したものを**構文の木** (syntax tree) と呼んでいる．すなわち，文をその構成部分に分割することによって，文の構造，構文を記述している．これから〈文〉は〈主部〉〈述部〉からなり，〈主部〉は〈冠詞〉

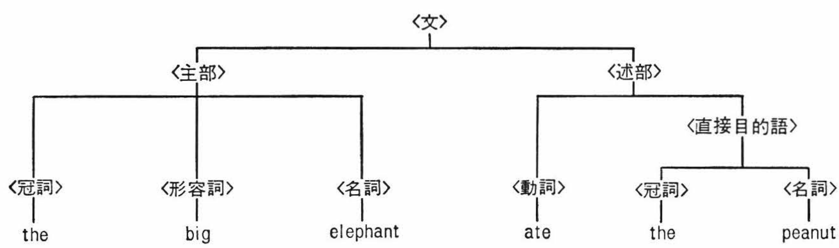


図 2.1　構文の木

〈形容詞〉〈名詞〉からなるなどということがわかる.

この構造を記述するために〈文〉のような新しい記号——“構文要素”あるいは“構文”のクラス——を使用した．新しい記号は，角かっこ“〈”および“〉”で括ることによって，言語の基本的な語と弁別できるようにしている.

われわれは,直観的にであれ,学校で習った文法を適用してであれ,“The big elephant ate the peanut”が，文であることがわかる．どんな簡単な英語の文に対しても形式ばらずに構文の木を作ろうと思えばそれは可能であろう．しかしながら，文を機械的に分解してゆくためには，この構文を示す，形式的かつ厳密な規則を与えなければならない．このために，**超言語** (metalanguage) を設計しよう．これは，他の言語を記述するための言語である．フランス語の授業の中で生徒にフランス語を説明するために英語がしばしば使われる．したがって，英語が超言語として使われている．英語の時間には，英語が英語自身を説明するための超言語として使われている.

ここでの主な関心は，プログラム用言語の**構文** (syntax) の記述であって，**意味** (semantics) の記述ではない．したがって，図2.1のような図式は，どんな意味をも伝えるものではない．このような構文の記述を**文法**(grammar) と呼ぶ.

これから用いる超言語は, Noam Chomsky(56)が最初に提案したもので，彼はこれによって自然言語の説明を試みた．ここで用いる特別な表記法は John Backus (59) によるものである．Chomsky がいうように，この超言語は，簡単な文からなる言語のほんの小さな部分集合の記述にしか役立たないので，言語学者たちは，もっと強力な超言語の方に向っている．ALGOL とか FORTRAN のように，かなり簡単な形式言語の構造を記述するのでさえこの超言語で十分であるというわけではない．しかしながら，この超言語が使用上，能力と実際を最もよく折中しているという理由で，ここで使用しよう.

もう一度“The big elephant ate the peanut”という文を考えよう．先に述べたように，図2.1は，〈文〉が〈主部〉〈述部〉からなっていることを示している．ここで，::=という記号で“左辺は右辺のものから成っている”という意味を表すことにすると，われわれの文法は，つぎのような規則を持っていることになる.

〈文〉　　::=〈主部〉〈述部〉
〈主部〉　::=〈冠詞〉〈形容詞〉〈名詞〉
〈冠詞〉　::= the

〈形容詞〉　::= big
〈述部〉　::=〈動詞〉〈直接目的語〉
〈動詞〉　::= ate
〈直接目的語〉　::=〈冠詞〉〈名詞〉
〈名詞〉　::= peanut
〈名詞〉　::= elephant

文法には，ある特定の構文要素の構成を記述するための規則が複数個あってもよいことに注意しよう．例えば，図2.1から〈名詞〉を構成しうるものを示すための規則が二つあることがわかる．

一たん規則の組が与えられると，これを用いてつぎのような機構によって文を**生成** (produce) あるいは**導出** (derive) することができる．(このため規則のことを**生成規則** (production) と呼ぶことが多い．) 構文要素〈文〉から始めて，〈文〉が ::=の左辺にある規則を探し，それを::=の右辺にある記号列で書換える．すなわち

〈文〉⇒〈主部〉〈述部〉

となる．このようにして，左辺の内容となっている記号列の一つを構文的に等価なものとし，置換えを行っているのである．この過程を繰返す．記号列〈主部〉〈述部〉の中から構文要素を1個，例えば〈主部〉を取出して，〈主部〉が::=の左辺である規則を探し，〈主部〉を::=の右辺の対応する記号列と置換える．これによって

〈主部〉〈述部〉⇒〈冠詞〉〈形容詞〉〈名詞〉〈述部〉

が生成される．記号⇒は，⇒の左辺の記号の一つから，文法中の規則の一つを使って⇒の右辺にある記号列を生成することを意味する．“The big elephant ate the peanut”という文の導出を完全に示すとつぎのようになるであろう．

〈文〉⇒〈主部〉〈述部〉
　⇒〈冠詞〉〈形容詞〉〈名詞〉〈述部〉
　⇒ the〈形容詞〉〈名詞〉〈述部〉
　⇒ the big〈名詞〉〈述部〉
　⇒ the big elephant〈述部〉
　⇒ the big elephant〈動詞〉〈直接目的語〉
　⇒ the big elephant ate〈直接目的語〉

⇒ the big elephant ate 〈冠詞〉〈名詞〉
⇒ the big elephant ate the 〈名詞〉
⇒ the big elephant ate the peanut

文のこのような導出を，新しい記号$\dot{\Rightarrow}$を用いて，縮めることにしよう．

〈文〉 $\dot{\Rightarrow}$ the big elephant ate the peanut

上の置換えの各段ではどの構文要素を置換えることも可能である．上の導出では，つねに最も左側の構文要素を置換えた．〈文〉 ::= 〈主部〉〈述部〉 のような規則を使っていろいろな文をたくさん記述することができることも注目しよう．そのためには，〈主部〉や〈述部〉を形成する方法がたくさんある必要がある．つぎの7個の規則

〈 文 〉 ::= 〈主部〉〈述部〉
〈主部〉 ::= We
〈主部〉 ::= He
〈主部〉 ::= I
〈述部〉 ::= ran
〈述部〉 ::= sat
〈述部〉 ::= ate

から，全部で9個の文を作ることができる

We ran	He ran	I ran
We ate	He ate	I ate
We sat	He sat	I sat

文法の目的の一つは，言語の全ての文を，適当な数の規則で記述することである．これは，言語というものは普通，無限個の文を含んでいることを悟るならば，意義のあることである．

これまで述べたことで，文法や言語の形式的な概念を記述する準備がほとんどできた．次節では，まず，われわれが普段定義なしで使っているいくつかの術語を定義することから始めなければならない．

演習問題　2.1

1. 標準的な辞書で，“language”，“meta”，“metalanguage”，“syntax” “semantics” という語を引いてみよ．

2. “John ate the big peanut”, “John ate the big brown peanut” “John ate the big brown roasted peanut”, “John ate the salted big brown roasted peanut” という四つの文の構文の木を描いてみよ．英語の単語は，形容詞をいくつでも重ねて修飾することができる．そこで，〈名詞〉を含んでその前にいくつでも形容詞をつけることのできる新しい構文要素（〈名詞句〉としよう）を2個の規則だけで与えてみよ．

2.2　記号と記号列

ここでは，ある基本的な“語彙”から持ってきた，語あるいは記号からなる全ての系列の集合の部分集合として，言語というものを，非公式に定義する．これらの系列には，どんな意味も与えない．例えば，英語は，語（if, he, is など）と句読点（コンマ，ピリオド，かっこなど）の系列からなる文から構成されている．プログラム言語 ALGOL は，**if**, **begin**, **end**, その他の区切り記号，英字，数字のような記号の系列であるプログラムで構成されている．偶数という言語は，0, 1, …, 9 という数字の系列のうち最後の桁が 0, 2, 4, 6 あるいは8であるような系列から成っている．

アルファベット (alphabet) とは空でない有限集合である。アルファベットの要素を**記号** (symbol) と呼ぶ．アルファベット A に属する記号からなる任意の有限系列を**記号列** (string) と呼ぶ．アルファベット $A=\{a, b, c\}$ のとき，a, b, c, ab, $aaca$ は，“このアルファベットの上の”記号列である．記号を1個も含まない空記号列 ε も記号列と認めることにする．記号列中における記号の順序は重要で，記号列 ab は，ba と同じものではないし，$abca$ は，$aabc$ とは違うものである．記号列 x の**長さ** (length) を $|x|$ と書いて，記号列中の記号の個数を表す．したがって

$$|\varepsilon| = 0,\ |a| = 1,\ |abb| = 3$$

などとなる．大文字 M, N, S, T, U, … はアルファベットに属する記号の変数あるいは名前として使用する．一方，小文字 t, u, v, w, …は記号列の名前として用いる．したがって

$$x = S\ T\ V$$

と書くと，“x は，記号 S, T, V がこの順に並んだ記号列である”ことを意味する．本書は，計算機を用いて書いている．計算機の出力では，添字がうま

く表せないので，ALGOL で使うかっこを用いて添字を表すことにする．したがって x[1], …, x[n] は，記号列 x_1, …, x_n を表す．混乱するおそれがないところでは，x[1], x[2], x[i], x[j], の代りに，[ ] をはずして，x1, x2, xi, xj と書いている．[†]

x と y が記号列であるとき，x のあとに y を続けて書いて得られた記号列を，**連接** (catenation) xy とする．例えば，$x = XY$, $y = YZ$ ならば，$xy = XYYZ$ であり，$yx = YZXY$ である．この約束に従えば，ε は，記号を1個も含まない記号列であるから，任意の記号列 x に対して，つぎのように書くことができる．

$$\varepsilon x = x\varepsilon = x.$$

もし $z=xy$ が記号列ならば，x を z の**頭部** (head)，y を z の**後部** (tail) と呼ぶ．もし y が空でなければ，x を**真の頭部** (proper head)，また x が空でなければ，y を**真の後部** (proper tail) と呼ぶ．したがって，$x = abc$ であれば，x の頭部は，ε, a, ab, abc であり，abc 以外は全て真の頭部である．

あるアルファベットの上の記号列の集合を，一般に，大文字 A, B, …で表す．A, B を記号列の集合とするとき，

$$AB = \{xy \mid x \in A,\ y \in B\}$$

で，二つの集合の**積** (product) を表し，これを x が A に属し，y が B に属するような，記号列 xy 全体の集合と読む．例えば，$A = \{a, b\}$ で，$B = \{c, d\}$ であれば，$AB = \{ac, ad, bc, bd\}$ となる．任意の記号列 x に対して $\varepsilon x = x\varepsilon = x$ であるから

$$\{\varepsilon\}A = A\{\varepsilon\} = A$$

が成立つ．ここで記号 ε を囲む { } の用法に注意しよう．$\{\varepsilon\}$ は，空記号 ε からなる集合を意味する．

つぎに，記号列の**べき** (power) を定義する．もし x が記号列ならば，x^0 は空記号列 ε であり，$x^1 = x$, $x^2 = xx$, $x^3 = xxx$, …一般に $x!n = \underbrace{xx\cdots x}_{n個}$ と定義する．ここで感嘆符！は，べきが変数の場合に用いる．計算機で

† **訳注**　訳書は計算機の出力ではないので理解を助けるために下付きの添字を使うが計算機のプログラムとの関係で原書のような記法を採ることもある．

は，上付きの添字は数字だけしか印刷できないからこのような記法を用いる．
$n>0$ であれば，

$x!n = xx!(n-1) = (x!(n-1))x$ である．ここで(　)は状況を明確にするために用いた超言語記号である．[†]

さらに，アルファベット A のべきもつぎのように定義することができる．

$$A^0 = \{\varepsilon\},\ A^1 = A,\ A^n = AA^{n-1}(\text{ただし}n>0)$$

これを用いて，本節の最後に二つの演算，すなわち集合 A の**閉包** (closure) A^* と**正の閉包** (positive closure) A^+ とを定義しよう．

$$A^+ = A^1 \cup A^2 \cup \cdots \cup A^n \cup \cdots$$
$$A^* = A^0 \cup A^+$$

したがって，$A = \{a, b\}$ ならば，A^* は，つぎのような記号列を持つ：

$$\varepsilon,\ a,\ b,\ aa,\ ab,\ ba,\ bb,\ aaa,\ aab\cdots$$

$A^+ = AA^* = (A^*)\ A$ に注意しよう．

例：

$z = abb$ とすると，$|z| = 3$. z の頭部は ε, a, ab, abb. また z の真の頭部は，ε, a, ab である．z の後部は，ε, b, bb, abb. また z の真の後部は ε, b, bb である．

$x = a$, $z = abb$ とすると
$zx = abba$, $xz = aabb$.
$z^0 = \varepsilon$, $z^1 = abb$, $z^2 = abbabb$, $z^3 = abbabbabb$.
$|z^0| = 0$, $|z^1| = 3$,　$|z^2| = 6$,　$|z^3| = 9$.

$S = \{a, b, c\}$ とすると

$S^+ = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \cdots\}$
$S^* = \{\varepsilon, a, b, c, aa, ab, ac, \cdots\}$

記号列 xy について，後半の部分—すなわち y—に，実際には関心がない場合には，xy の代りに $x\ldots$ と書く方が便利な（そして普通はより意図が明白になる）ことがある．このように点を3個“...”並べることによって，空記

---

† **訳注**　訳書は計算機の出力ではないので読易さを考えて，べきを表すために感嘆符を用いることはせず，全て上付き添字を用いる．すなわち $x^n = xx^{n-1} = x^{n-1}x$ などと表す．

号列を含んだ任意の記号列を意味する．よく使われる用法は下記のようなものである．

記　法	意　味
$z = x\ldots$	x は記号列 z の頭部であり，z の後部には関心がない．
$z = \ldots x$	x は記号列 x の後部であり，z の頭部には関心がない．
$z = \ldots x \ldots$	x が記号列 z の適当な場所に現れる．
$z = S\ldots$	記号列 z の先頭の記号が S である．
$z = \ldots S$	記号列 z の最後の記号が S である．
$z = \ldots S\ldots$	記号列 z 中に記号 S がある．

演習問題　2.2

1. “記号列” および “連接” を定義せよ．
2. $A=\{\$\}$，$z=\$$ とするとき，つぎの記号列を書き下し，それらの長さをいえ：z，zz，z^2，z^5，z^0．A^* はどんなものか．
3. $A = \{0,\ 1,\ 2\}$，$x = 01$，$y = 2$，$z = 011$ とするとき，つぎの記号列を書き下せ，またそれらの長さ，頭部，後部をそれぞれ示せ：xy, yz, xyz, x^4，$(x^3)(y^2)$，$(xy)^2$，$(yxx)^3$
4. $A = \{0, 1, 2\}$ とする．このとき集合 A^+ および A^* の最も短い記号列を書き下せ．

2.3　文法および言語の形式的定義

いよいよ，ここで，規則（これは生成規則と呼ばれているもの）の概念を形式化し，これらの規則を用いる文法と言語を抽象的に定義しよう．

(2.3.1)　**定義**　　**生成規則**(production) あるいは**書換え規則**(rewriting rule) とは，順序対 (U, x) であり，一般に

$$U \ ::= \ x$$

と書く．ここで U は記号であり，x は有限個の記号からなる空でない系列である．U を生成規則の**左辺**(left part)，x を**右辺**(right part) と呼ぶ．生成規則の代りに**規則**というより短い言葉を使うことにしよう．■

(2.3.2)　**定義**　　文法 $G[Z]$ とは，規則の空でない有限集合である．Z は記号であって，少くとも 1 個の規則の左辺に現れていなければならない．この

記号を**出発記号** (distinguished symbol) と呼ぶ．規則の左右両辺に現れる全ての記号を併せて**語彙**(vocabulary)と呼ぶ．■

(2.3.3)　例　　文法 $G1$ [〈整数〉] をつぎの13個の規則とする．

(1)　〈整数〉 ::= 〈no〉
(2)　〈 no 〉 ::= 〈no〉〈数字〉
(3)　〈 no 〉 ::= 〈数字〉
(4)　〈数字〉 ::= 0
(5)　〈数字〉 ::= 1
(6)　〈数字〉 ::= 2
(7)　〈数字〉 ::= 3
(8)　〈数字〉 ::= 4
(9)　〈数字〉 ::= 5
(10)　〈数字〉 ::= 6
(11)　〈数字〉 ::= 7
(12)　〈数字〉 ::= 8
(13)　〈数字〉 ::= 9

V = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 〈数字〉, 〈no〉, 〈整数〉}.■

(2.3.4)　**定義**　　文法 G が与えられたとき，規則の左辺に現れる記号を**非終端記号** (nonterminal)または**構文要素** (syntactic entity) と呼ぶ．これらの記号の集合を V_N とする．V_N に属さない記号を**終端記号** (terminal symbol) と呼び，これらの記号の集合を V_T とする．■

この定義から $V = V_N \cup V_T$ である．非終端記号を終端記号と区別するために，角かっこ“〈”と“〉”を用いるのが普通である．例2.3.3の文法 $G1$ では，0, 1, 2, 3, 4, 5, 6, 7, 8, 9 が終端記号である．また〈数字〉，〈no〉，〈整数〉が非終端記号である．

同じ非終端記号が同じ規則の別の場所に現れて，それぞれを区別する必要があるときには，上付きの添字を用いる．したがって〈no〉 ::= 〈no〉〈数字〉に対して〈no^1〉 ::= 〈no^2〉〈数字〉のような書き方ができる。

同じ左辺を持った規則の集合 U ::= x, U ::= y, …, U ::= z を縮めて

$$U ::= x \mid y \mid \cdots \mid z$$

と書く．例えば，文法 $G1$ では

〈整数〉　::= 〈no〉
〈 no 〉　::= 〈no〉〈数字〉|〈数字〉
〈数字〉　::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

と書くことができよう．

この表現法を特に**BNF**と呼んでいる．これは，**Backus-Normal Form** あ

るいは **Backus-Naur Form** の頭文字をとったものである．これは，ALGOL 60の報告書の中で ALGOL を記述するために，Backus(59)によって最初に展開された [Naur(63a)参照]．Naur は，ALGOL 報告書の編者である．形式言語を記述するために用いられる表記法はほかにいくつかあるが，これは後で述べよう．

文法が与えられたならば，この文法に対応する言語はどのように示したらよいのだろう．そしてこの言語の文とはどんなものか．このためには，2.1 節で直観的に用いてきた記号⇒および$\stackrel{+}{\Rightarrow}$を定義しなければならない．$v \Rightarrow w$ の非公式な意味は，v 中の非終端記号の一つをそれを左辺とする規則の対応する左辺で置換えて v から w を導出することができるということである．

(2.3.5)　**定義**　　G を文法とする．記号列 x, y が存在し，$U ::= u$ が G の規則のとき，記号列 v, w に対して

$$v = xUy,\ w = xuy$$

と書けるならば，v は w を**直接生成する**(directly produce)といい

$$v \Rightarrow w$$

と書く．このことを，w は v の**直接導出** (direct derivation) である，あるいは，w は v に**直接還元**される(directly reduce)ともいう．■

記号列 x および y は空であってよい．それ故，文法 G の任意の規則 $U ::= u$ に対して直接生成 $U \Rightarrow u$ がある．図2.2は，例2.3.3の文法 $G1$ 上で定義した記号を用いて，直接導出の例をいくつか示したものである．

(2.3.6)　**定義**　　直接導出の系列

$$v = u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n = w \qquad (n>0)$$

が存在するならば，v は w を，**生成する** (produce) あるいは，w を v に**還元する**(reduce)といい，$v \stackrel{+}{\Rightarrow} w$ と書く．この系列を**長さ** (length) n の導出という．記号列 w は，v に対する**語**(word)ともいわれている．なお，もし $v \stackrel{+}{\Rightarrow} w$ または $v = w$ ならば $v \stackrel{*}{\Rightarrow} w$ と書く．■

導出の例を示そう．図 2.2 の第1行を見てもらいたい．これは〈整数〉⇒〈no〉となり，導出の長さは1である．つぎに2行目に移ると，これは〈整数〉$\stackrel{+}{\Rightarrow}$〈no〉〈数字〉を示しているので，導出の長さは2である．このように図2.2を下にたどると，

v		w	使用した規則	x	y
<整数>	⇒	<no>	1	ε	ε
<no>	⇒	<no><数字>	2	ε	ε
<no>　<数字>	⇒	<数字><数字>	3	ε	<数字>
<数字><数字>	⇒	2<数字>	6	ε	<数字>
2<数字>	⇒	22	6	2	ε

図 2.2　直接導出の例

〈整数〉 ⇒ 〈no〉 ⇒ 〈no〉〈数字〉 ⇒ 〈数字〉〈数字〉
⇒ 2 〈数字〉 ⇒ 22

したがって〈整数〉$\stackrel{+}{\Rightarrow}$22 となり導出の長さは5である．
記号列中に非終端記号が存在するかぎり，それから新しい記号列が導出される可能性があることに注意しよう．しかしながら，もし非終端記号がなくなれば，導出は**終了** (terminate) しなければならない．したがって規則の左辺に現れることのない記号を**終端記号** (terminal symbol) と名付けるのである．

G[〈整数〉] という文法によって記述された言語はどんなものと考えたらよいのか．つぎの定義によって，この言語は1桁以上の数字の系列の集合であることがわかる．

(2.3.7)　**定義**　$G[Z]$ を文法とする．もし記号列 x が，出発記号 Z から導出できる――$Z \stackrel{*}{\Rightarrow} x$――ならば，$x$ を**文形式** (sentential form) と呼ぶ．**文** (sentence) とは，終端記号のみからなる文形式である．言語 $L(G[Z])$ とは，つぎのような文の集合である．

$$L(G) = \{x \mid Z \stackrel{*}{\Rightarrow} x \text{ かつ } x \in V_T^{+}\} ■$$

したがって，言語とは，全ての終端記号列――集合 V_T の上の記号列――の集合の部分集合である．文の構造は，文法によって与えられる．しかし，いくつかの別々の文法から，同一の言語が生成されることもありうる．“文法によって定義された言語の文”という代りに，しばしば“文法の文”ということがある．

(2.3.8)　**例**　われわれにとってなつかしい文 “the big elephant ate the peanut” を変形した文を含む言語の文法を書いてみる．文法 $G2$[〈文〉] はつぎのとおりである：

〈文〉　::= 〈主部〉〈述部〉
〈主部〉　::= 〈冠詞〉〈形容詞〉〈名詞〉|〈代名詞〉
〈述部〉　::= 〈動詞〉〈直接目的語〉
〈直接目的語〉 ::= 〈冠詞〉〈名詞〉
〈代名詞〉　::= he
〈冠詞〉　::= the
〈形容詞〉　::= big
〈動詞〉　::= ate
〈名詞〉　::= elephant | peanut

終端記号の集合は{he, the, big, ate, elephant, peanut}である．言語 $L(G2)$ は，終端記号の系列の集合で出発記号〈文〉から導出できる（出発記号〈文〉に対する語である）．文にはつぎのようなものがある：

he ate the peanut
he ate the elephant
the big elephant ate the elephant ■

つぎの定義は重要である．読者は，この定義が理解できたと確信できるまでは，先に進んではならない．

(2.3.9)　**定義**　$G[Z]$を文法，$w = xuy$ を文形式とする．このとき

$$Z \overset{*}{\Rightarrow} xUy \text{ かつ } U \overset{+}{\Rightarrow} u$$

ならば，u は，非終端記号 U に対する文形式 w の**句**(phrase)と呼ばれる．さらに，

$$Z \overset{*}{\Rightarrow} xUy \text{ かつ } U \Rightarrow u$$

であれば，u は，**単純句**(simple phrase)と呼ばれる．■

句という術語は，注意して使わなければならない．$U \overset{+}{\Rightarrow} u$ だからといって，必ずしも，u が文形式 xuy の句であるという意味ではない．さらに $Z \overset{*}{\Rightarrow} xUy$ であることが必要である．これを説明するために，例2.3.3の文法 $G1$ における文形式〈no〉1 を考えてみよう．〈整数〉::=〈no〉という規則があれば，〈no〉は句になるだろうか．絶対に否である．なぜならここでは，出発記号

〈整数〉から〈整数〉1 を生成できないからである．では〈no〉1 の句は，一体どれであろう．

$$〈整数〉 \Rightarrow 〈no〉 \Rightarrow 〈no〉〈数字〉 \Rightarrow 〈no〉 1$$

であるから，

（1）〈整数〉 $\stackrel{*}{\Rightarrow}$ 〈no〉 かつ 〈no〉 $\stackrel{+}{\Rightarrow}$ 〈no〉 1

（2）〈整数〉 $\stackrel{*}{\Rightarrow}$ 〈no〉〈数字〉 かつ 〈数字〉 $\stackrel{+}{\Rightarrow}$ 1

したがって，〈no〉1 と 1 が句である．単純句は 1 だけである．

われわれは，今後，文形式の**最も左側の** (leftmost，最左) 単純句にしばしば言及しなければならないので，これに名前をつけておこう．

(2.3.10)　**定義**　　任意の文形式において最も左側の単純句を**把手** (handle) と称する．■

例 2.3.3 の文法 $G1$ は，無限の言語を記述している．すなわち，この言語には無限個の文が存在する．なぜなら規則〈no〉::= 〈no〉〈数字〉の両辺に，〈no〉が現れているからである．ある意味で，〈no〉は自分自身を定義するために用いられている．一般に，もし $U \stackrel{+}{\Rightarrow} \cdots U \cdots$ であるならば，文法は U に関して**再帰的** (recursive) であるという．もし，$U \stackrel{+}{\Rightarrow} U \cdots$ ならば，**左再帰的** (left-recursive)，もし $U \stackrel{+}{\Rightarrow} \cdots U$ ならば，右再帰的という．規則が $U ::= U \cdots$ ($U ::= \cdots U$) の形式であれば，その規則は，左再帰的 (右再帰的) であるという．もし言語が無限であるならば，その言語を定義している文法は，再帰的でなければならない．

演習問題　2.3

1. 文法 $G[〈id〉]$ の規則はつぎのようなものである

   〈id〉 ::= a | b | c | 〈id〉 a | 〈id〉 c | 〈id〉 0 | 〈id〉 1

   V_T と V_N を書け．つぎの記号列が導出可能ならば，それぞれ導出を示せ：a，$ab0$，$a0c1$，$0a$，11，aaa.
2. 例 2.3.3 で与えた文法 $G1$ に対する言語 $L(G1)$ の文はどんなものか.
3. 偶数の集合という言語を記述する文法を書け.
4. 偶数の集合という言語の文法を書け．ただし，頭につづくゼロがあっては

ならない.

5. 文法 G の規則を $\langle A\rangle ::= b\ \langle A\rangle | cc$ とする. cc, bcc, $bbcc$, $bbbcc$, ・・・ が $L(G)$ に属することを示せ.

6. 言語$\{ab^na \mid n=0,1,2,\cdots\}$ に対する文法を構成せよ.

7. 言語$\{a^nb^n \mid n=1,2,3,\cdots\}$ に対する文法を構成せよ.

8. つぎのような文法 $G3$[〈式〉] は2項演算子を用いる算術式にしばしば用いられる(ここで i は "名前(identifier)" を表す).

〈 式 〉 ::= 〈項〉|〈式〉 + 〈項〉|〈式〉 − 〈項〉
〈 項 〉 ::= 〈因子〉|〈項〉 * 〈因子〉|〈項〉 / 〈因子〉
〈因子〉 ::= (〈式〉)| i

つぎの算術式の導出を与えよ: i, (i), $i*i$, $i*i+i$, $i*(i+i)$

9. 文法 $G3$ (問題8)の文形式 〈式〉 + 〈項〉 * 〈因子〉 の句と単純句を全て列挙せよ.

2.4 構文の木とあいまいさ

構文の木 (syntax tree) は文の構文を理解するのに役立つ. 図2.1は構文の木の例である. 例2.3.3の文法 $G1$ の文22の導出

(2.4.1) 〈整数〉 ⇒ 〈no〉 ⇒ 〈no〉〈数字〉
⇒ 〈数字〉〈数字〉 ⇒ 2〈数字〉 ⇒ 22

から木を描く方法を説明しよう. **図2.3a** に示すように, 出発記号〈整数〉から始めて, **枝**(branch) を下向きに出しこれで最初の直接導出を示す. 枝とは**節点**(node)を下に伴った線の集合である. 最初の直接導出〈整数〉⇒〈no〉においての枝の節点を並べたものがこの枝の名前と置換わる記号列を形成する. つぎの導出を示すために, 上で置換えられた記号に対応する節点から枝を描きその先に付いた節点の列が, つぎに置換わる記号列を構成する. (**図2.3b**). 順次このように続けて, **図2.4**のような3段階の構文の図式が得られる.

構文の木の**端点** (end node) とは, もはやその節点から下に枝が出ていない節点をいう. このような端点を左から右に眺めると, 今の導出で生成された記号列を形成しており, これこそ木が表しているものである. したがって, (2.4.1) の3番目の直接導出を行うと, 端点の記号列は〈数字〉〈数字〉となって

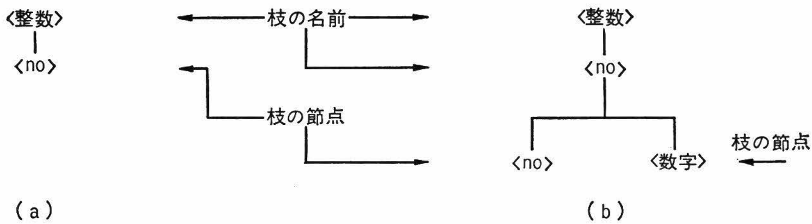


図2.3　2段階の導出に対する構文の木

いる(図2.4a 参照).

端枝 (end branch) とは，その枝についた節点が全て端点であるような枝をいう．図2.3bには端枝が1本ある，すなわち2個の端点〈no〉と〈数字〉がついている．図2.4cには，端枝が2本あり，ともに，2という端点がついている．つぎにあげる術語は木を議論するときにしばしば用いられるものである．Nを木の節点としよう．Nの**息子**(son)とは，Nと名付けた枝についた節点をいう，Nはこの節点の**父**(father)である．息子達は互に**兄弟**(brothers)であって，一番左側の息子が，最も年下である．図2.4bで，〈no〉は〈整数〉の一人息子であり，この息子が二人の息子〈no〉と〈数字〉を持っている．節点2の父は〈数字〉である．

構文の木の**部分木** (subtree) とは，構文の木の上にある節点（これをこの部分木の**根**(root)と称する）とその節点から下に出ている構文の木の一部分（がもしあれば）を併せたものをいう．部分木は，句と関連が強く，部分木の端点を並べるとその部分木の根に対する句を形成している．これを少し詳しくみていこう．

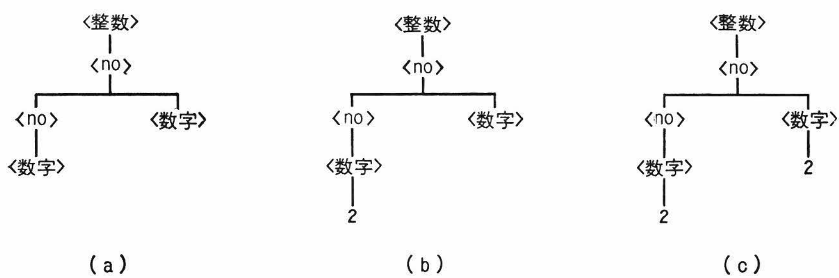


図2.4　導出(2.4.1)に対する構文の木

もし U が部分木の根であり，u がこの部分木の端点を並べた記号列であるならば，$U \overset{*}{\Rightarrow} u$ が成立つ．x が u の左にある記号列，y が u の右にある記号列とすると，xuy は，もとの木が表す文形式である．$Z \overset{*}{\Rightarrow} xUy$ とこれで xuy の中で u が U に対する句であることを意味している．

構文の木から導出を作る

今の過程を逆にたどれば，構文の木から導出を再構成することができる．図2.4cの端点から，最終的な記号列が22であることがわかる．最も右側の端枝が，

2〈数字〉⇒ 22

という直接導出を示している．2〈数字〉に対する構文の木に至るために，この枝を，木から**切取る** (prune)．この枝を図2.4cから切取ると図2.4bが得られる．この過程を，**直接還元**と呼ぶことが多い．

図2.4bをみると，この図に至った直接導出は〈数字〉〈数字〉⇒ 2〈数字〉でなければならないことがわかる．すなわち，ここまでで

〈数字〉〈数字〉⇒ 2〈数字〉⇒ 22

がわかった．このようにつねに構文の木の端枝の一つに対応するその前の直接導出を再構成しその枝を切取ることを続けてゆく．

構文の木について，つぎのように要約しよう：

- □ 全ての構文の木に対して，それぞれ導出が少くとも一つ存在する．
- □ 全ての導出に対して，それぞれに対応する構文の木が一つ存在する（しかし，いくつかの導出が同じ木に対応することもある）．
- □ 木の枝は直接導出を示す．そこでは，その枝の名前が，その枝についた節点と置換っている．したがって，この文法には，左辺が枝の名で右辺が，この枝についた節点を並べた記号列である規則が存在する．
- □ 木の端点を並べると導出された文形式を形成する．
- □ U を文形式 $w = xuy$ に対応する部分木の根としよう．ただし u は，この部分木の端点を並べた記号列とする．すると u は，文形式 w の U に対する句である．もし部分木が1本の枝であれば，u は，単純句である．

文法 $G1$ の文22には(2.4.1)の導出以外にもつぎのような導出がある：

(2.4.2)　〈整数〉⇒〈no〉⇒〈no〉〈数字〉
⇒〈数字〉〈数字〉⇒〈数字〉2 ⇒ 22

この構文の木は(2.4.1)の導出の木と同様である．実際，22には，さらにほかの導出もある：

(2.4.3)　〈整数〉⇒〈no〉⇒〈no〉〈数字〉
⇒〈no〉2 ⇒〈数字〉2 ⇒ 22

このような導出はみな，導出の過程で，適用する規則の順番が違っているにすぎないこと，そして，構文の木は，導出が生ずる順序を正確に示しているものではないことに注意しよう．このような導出における順序の違いは，現時点では，実際上重要ではないので，同じ木を作りだす導出は同等なものとみなすことにする．

あいまいさ

より重要なことは，構文の木の一意性である．つぎのような規則を含んだ文法を考えよう(この文法にはこのほかにも規則がある)：

〈文〉	::=	〈主部〉〈述部〉
〈文〉	::=	〈述部〉〈直接目的語〉
〈主部〉	::=	〈名詞〉
〈直接目的語〉	::=	〈名詞〉
〈述部〉	::=	〈動詞〉
〈名詞〉	::=	time｜flies
〈動詞〉	::=	time｜flies

これらの規則を用いると“time flies”という文を，二様の方法で生成することができる．これには二様の構文の木が対応する(図2.5)．
いずれの場合も英語として意味が通じる．すなわち一方は“time flies by very quickly(時はどんどん流れる)”，他方は“go find out how fast flies fly (蝿がどんなに速くとぶか測ってみよ)”である．文脈をぬきにしては，この文を正しく“構文解析”することができない，すなわち，この文をあいまいさをなくして，その構成部分に分解することができないので，この文の意味しているところを述べることができないことに問題がある．もし，コンパイラは正しい原始プログラムを全て翻訳することができなければならないというものであれば，

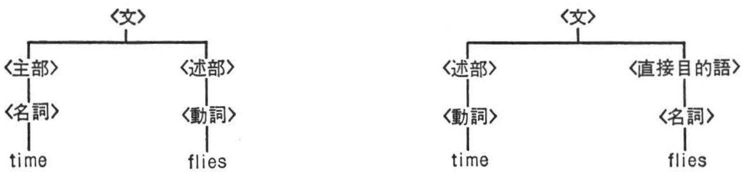


図 2.5 あいまいな文に対する構文の木

あいまいさのないように言語が定義されていることを求めるのが理に適っている．したがってつぎの定義を導入しよう．

(2.4.4) **定義** もし文法のある文に構文の木が2通り存在すれば，その文は**あいまいである**(ambiguous)という．あいまいな文を生成する文法は**あいまいである**という．さもなければ，**あいまいでない** (unambiguous) という．■

ここであいまいであるといっているのは文法についてであり言語についてではないことに注意しよう．あいまいな文法を変更することによって――もちろん文はそのままにして――全く同じ文の集合に対して，あいまいさのない文法を作ることができることがある．しかしながら，あいまいさのない文法が存在しないような言語もあり，このような言語は，**本質的にあいまいである** (inherently ambiguous) と称している(2.10節参照)．

ここでは，構文だけを問題にしていることを思い起そう．上の定義にしたがってある文があいまいでないとしても，語の意味があいまいなために，その文が何を意味しているのかなお不明なこともありうる．

不幸にして，あいまいさの**性質は決定不能** (undecidable) であることが証明されている．つまり，任意の BNF 文法を受理し有限の時間内で，これがあいまいであるかないかをはっきりと決定するアルゴリズムが存在しないという意味である．われわれにできるのはかなり単純ではあるが自明ではない条件を展開することで(第5，6章)，ある文法がこれらの条件を満足するならば，その文法があいまいでないことを保証してくれるものである．これらは，無あいまいさに対する十分条件であって，決して必要条件ではない．

算術式に対するあいまいでない文法

もし文形式があいまいであると，構文の木がいくつもでき，したがって，一般には，把手がいくつもできることになる．これを例で説明しよう．この例



図 2.6　⟨E⟩ + ⟨E⟩ * ⟨E⟩ に対する2本の構文の木

は，算術式を表すのに使える文法を提供してくれる．被演算子は i（名前を表す）だけで，かっこと2項演算子 ＋ と ＊ 用いた算術式に対する文法

(2.4.5)　　⟨E⟩ ::= ⟨E⟩ + ⟨E⟩ | ⟨E⟩ * ⟨E⟩ | (⟨E⟩) | i

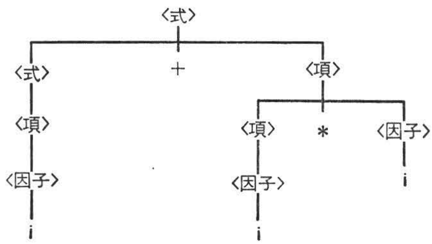
を考えよう．文形式 ⟨E⟩ + ⟨E⟩ * ⟨E⟩ には，構文の木が2本（図 2.6）と把手が2個——⟨E⟩ + ⟨E⟩ と ⟨E⟩ * ⟨E⟩——存在する．もしこの文形式の構文解析をしようとするならば，どの把手を還元すればよいのだろう．この文法にはあいまいさがあり，どちらの把手にしてもよい．これでは，乗算を加算の前に行うのか後に行うのか決められない．

つぎに，演習問題 2.3.8 の文法 $G3$[⟨式⟩] を考えてみよう：

(2.4.6)　　⟨式⟩　::=　⟨項⟩ | ⟨式⟩ + ⟨項⟩ | ⟨式⟩ − ⟨項⟩
　　　　　　⟨項⟩　::=　⟨因子⟩ | ⟨項⟩ * ⟨因子⟩ | ⟨項⟩/⟨因子⟩
　　　　　　⟨因子⟩　::=　(⟨式⟩) | i

式 $i+i*i$ に対する木はただ一つで次ページに示すようになる．したがって，この文法によればこの文はあいまいではない．事実，$G3$ の全ての文はあいまいでない．$G3$ にしたがって，$i+i*i$ で乗算と加算のどちらを先に行うかを決定してみよう．次図の木で ＋ の被演算子は，i を生成することになる ⟨式⟩ と $i*i$ を生成することになる ⟨項⟩ とである．これから加算のための ⟨項⟩ を作り出すために $i*i$ の乗算を先に行わなければならないという意味を汲みとることができる．

文法 $G3$ が文法(2.4.5)より好ましいのは二つの理由による：すなわち，この文法はあいまいでなく，乗算が加算に優先することを示しているからである．



演習問題　2.4

1. 下記の導出に対して，それぞれ構文の木を描け．

 a. 〈整数〉⇒〈no〉⇒〈no〉〈数字〉⇒〈no〉3⇒〈no〉〈数字〉3⇒〈no〉23
 ⇒〈数字〉23⇒123

 b. 〈整数〉⇒〈no〉⇒〈no〉〈数字〉⇒〈no〉3⇒〈no〉〈数字〉3
 ⇒〈数字〉〈数字〉3⇒1〈数字〉3⇒123

 a，**b** はともに同じ構文の木になるか．

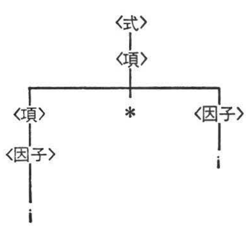
2. 下記の導出に対して，それぞれ構文の木を描け．

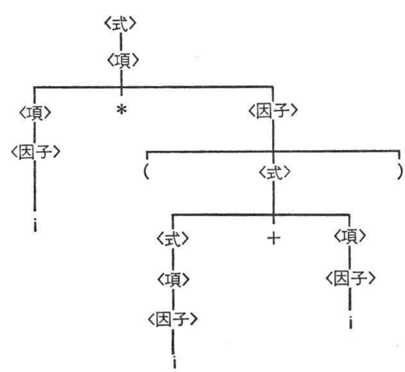
 a. 〈式〉⇒〈項〉⇒〈因子〉⇒ i

 b. 〈式〉⇒〈項〉⇒〈因子〉⇒(〈式〉)⇒(〈項〉)⇒(〈因子〉)⇒(i)

 c. 〈式〉⇒＋〈項〉⇒＋〈因子〉⇒＋i

3. 下記の構文の木からそれぞれ導出を作れ．





4. 下記の文法 G[〈式〉] はあいまいである．これを，文 $i+i*i$ と $i+i+i$ に対してそれぞれ二つずつ構文の木を描いて示せ．

　　〈式〉::＝ i|(〈式〉)|〈式〉〈op〉〈式〉
　　〈op〉::＝ ＋|－|＊|/

5. 文法 $G3$(2.4.6) の二つの文 $i+i*i$ と $i+i+i$ があいまいでないことを示せ．$i+i*i$ でどちらの演算子が優先するか．また $i*i*i$ ではどちらの演算子が優先するか．

2.5　構文解析問題

文形式の**構文解析** (parse) とは，導出の組立て，あるいはその構文の木の組立てをいう．構文解析用プログラム，または**構文解析部**(parser)は問題としている文法の文しか認識しないので，**認識部**(recognizer)とも呼ばれている．もちろん，現下の問題は，プログラム用言語のプログラムを認識しようというものである．

ここで説明する構文解析のアルゴリズムは全て，**左から右への**(left-to-right)構文解析アルゴリズムである．なぜなら，このアルゴリズムでは，問題の記号列の最も左にある記号をまず調べて処理し，必要なときしか右に進まないからである．右から左への構文解析も同様に定義できよう．しかし左から右の方がずっと自然である．プログラムの実行は，左から右になされるし，読む順序も左から右である．

構文解析アルゴリズムは，さらに**下向き解析**(top-down)と**上向き解析** (bottom-up)という二つの範疇に分類される．この二つの術語は，構文の木が組立てられる方法を示している．下向き解析法では，木を根――出発記号――から始めて，下向きに端点に向けて組立てる．整数に対する文法

(2.5.1)　　N ::＝ D|ND
　　　　　D ::＝ 0|1|2|3|4|5|6|7|8|9

について文35を考えてみよう．第1段は，図2.7の最初の木に示すように直接導出 N ⇒ ND を作ることになるだろう．以後の各段では，その時の文形式 xVy における最も左側の非終端記号 V が，規則 V::＝u の右辺と置換えられて，次の段の文形式になる．これを文35について図2.7の5本の構文の木で示した．もちろん，最初の記号列35に立到るように工夫してある．

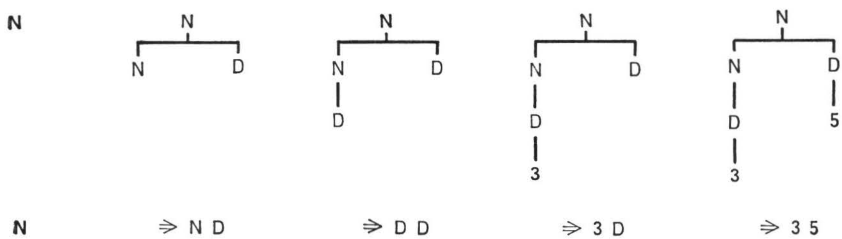


図2.7 下向き構文解析とそれによる導出

上向き構文解析法は，与えられた記号列から出発して，それを還元して出発記号に至る方法である．解析の第1段は，3をDに還元して，文形式D5作るものである．これで，図2.8で最も右側にある部分的な構文の木に見られるように，直接導出 D5 ⇒ 35 を作った．次の段は，図2.8の右から2番目の木に示すように，DをNに還元するものである．このようにして，図の左端の木が組上るまで続ける．下図の木を右から左に並べたのは，でき上った導出がはっきりわかって，それを自然に左から右に読めるからである．二つの異なった解析法で生成された導出はそれぞれ別のものであるが，生成された構文の木は同じものであることに注意しよう．

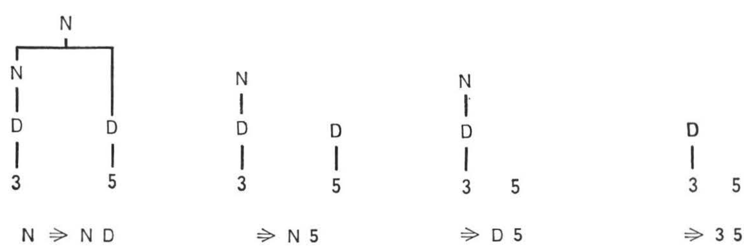


図2.8 上向き構文解析とそれによる導出

以後，左から右への上向き構文解析によって生成される導出の型を参照することが多いので，これに対する術語を決めておこう．このような構文解析の各段では，そのときの文形式の把手（最も左側の単純句）が還元され，したがって，**把手の右にある記号列は，つねに終端記号のみから成っている．**

(2.5.2) **定義**　直接導出 $xUy \Rightarrow xuy$ において，記号列 y が終端記号のみから成っているならば，この直接導出を**正準である** (canonical) といい，

$xUy \underset{c}{\Rightarrow} xuy$ と書く．導出 $w \overset{+}{\Rightarrow} v$ において，各段の直接導出が全て正準であれば，この導出を正準であるといい，$w \underset{c}{\overset{+}{\Rightarrow}}$ と書く．■

全ての文には，正準な導出があるが，文形式には必ずしもあるわけではない．文法 (2.5.1) で，文形式 3D を例に考えてみよう．この導出は

$$\langle 整数 \rangle \Rightarrow \mathrm{ND} \Rightarrow \mathrm{DD} \Rightarrow 3\mathrm{D}$$

だけであり，2番目と3番目の直接導出は，正準ではない．これで困ることはない．何故ならば，われわれの主な関心は，プログラムの構文解析であり，プログラムとは，プログラム用言語の文であるからである．正準な導出を持つ文形式は，**正準な**文形式と呼ばれる．

これまで，構文解析におけるつぎのような重要な問題を棚上げしてきた．

1. 下向き構文解析で，置換えの対象となる，最も左側の非終端記号を V とする．V につぎのような n 個の規則があるとしよう．

$$V ::= x_1 | x_2 | \cdots | x_n$$

　このとき V と置換える記号列 x_i をどのようにして知るのか．

2. 上向き構文解析で，把手は，各段で還元されるべきものである．この把手をどのようにして見出すのか，それを何に還元するのか．

これらの問題は，必ずしも容易には答えられない——演習問題2でこれを強調してある．第4章で下向き構文解析法に関する問題の解決を行い，第5章と第6章で，上向き構文解析法の問題を解決するためのさまざまな技法について議論する．

解決法の一つは，可能な選択枝の中からランダムに一つを拾いあげ，望むらくはそれが正しいものとしておく．後になって誤りであることが検出されたら，誤りの原因となった点まで戻って，別の選択枝をとりあげて試みる．これを**逆戻り** (backup) と称する．この解法は，非常に時間がかかることは明らかである．

別の解決法は，現に処理している部分記号列の付近の文脈を眺めることである．これは，A+B*C のような算術式を評価するとき，われわれが普通に行っているやり方である．A+B をみるとこれを先に評価してよいかどうか自らに問い，A+B のあとに * が続くので，先に評価してはならないと自ら答えてい

るのである.

構文解析アルゴリズムを実際に用いてコンパイラを書いた最初のものの一つが Irons (61 a) に述べられている. Ingerman (66) の構文向きトランスレータもこのアルゴリズムに基いている. この方法は, 上向き解析法とも下向き解析法ともいえず, 両者を混成したものである. これについてはこれ以上議論しない.

演習問題 2.5

1. 例 2.3.3 の文法 $G1$ の文 561, 0, および 0012 について, それぞれ正準な構文解析列を示せ.

2. 下記の文法の文 (,)(*, i(, (+(, (+(i(, および (+)(i(* i(の構文解析列を示せ.

$$\begin{array}{lll} \langle\text{goal}\rangle & ::= & V1 \\ V1 & ::= & V2 \mid V1 \ \ i \ \ V2 \\ V2 & ::= & V3 \mid V2 \ + \ V3 \mid i \ \ V3 \\ V3 & ::= &) \ \ V1 \ * \mid (\end{array}$$

3. 文法 $V ::= aaV \mid bc$ の言語は何か.

2.6 文法におけるいくつかの関係

関　係

2.3 節で定義した ⇒ や $\stackrel{+}{\Rightarrow}$ などの記号は, 記号列の間の関係を示す例である. 一般に, ある集合の上の (2項) 関係とは, その集合の2個の記号の任意の順序対について任意の性質が成立するかしないかをいう.

関係の別の例に, 整数の集合の上で定義された, **より小さい** (<) がある: $j-i$ が正の整数でそのときに限り, $i<j$ が成立つ. 数学的でない関係の例では人間の集合に対して, **～は～の子供である**がある. 人 A は人 B の子であるか, そうでないかのどちらかである.

関係を表すのに, 挿入記法(インフイクス)を用いる. すなわち集合の要素 c と d について, 関係 R が成立すれば cRd と書く. (ここで, 2要素の順序が重要で, cRd から自動的に dRc とはならないことに注意.) また, 関係とは, その関係が成立つような順序対の集合であると考えることができる. すなわち, cRd が成立ちかつそのときに限り $(c,d)\in R$.

(c,d) が集合 R の要素であることが集合 P の要素でもあることを意味しているならば，関係 P は関係 R を**含む**という．

関係 R の**逆**を TRANSPOSE(R) と書いて，dRc が成立ちそのときに限り

$$c \quad \mathrm{TRANSPOSE}(R) \quad d$$

と定義する．

関係**より大きい**（>）の逆は**より小さい**（<）となる．**〜の子供である**の逆は**〜の親である**となる．

関係 R について，集合の全ての要素 c に対して cRc が成立つとき，R は**反射的** (reflexive) であるという．例えば関係**より小さいか等しい**（≤）は反射的である（全ての実数について $i \leq i$ である）．一方関係**より小さい**は，反射的でない．

関係 R について，aRb かつ bRc ならば aRc が成立つとき，R は**推移的** (transitive) であるという．整数の集合に対して**より小さい**は推移的である．**〜の子である**は推移的ではない．もし太郎が次郎の子で，次郎が三郎の子であれば，普通は，太郎は三郎の子ではない．

任意の関係 R が与えられたとき，それから**推移的閉包** (transitive closure) R^+ と呼ばれる，新しい関係を定義したい．このように呼ばれるのは，この関係が，R を含む任意の推移関係に含まれるからである．いいかえれば，もし P が推移的でかつ R を含む——すなわち (c,d) が R に属するならば P にも属する——ならば，P は R^+ も含む．

まず，同一の集合の上で定義された二つの関係 R および P が与えられたとき，集合の要素 e が存在して，cRe および ePd が成立ちそしてその時に限り

$$c \quad RP \quad d$$

と書いて，二つの関係 R, P の**積** (product) と呼ばれる新しい関係を定義する．この演算は任意の関係，P, Q, R に対して結合律——$R(PQ)=(RP)Q$——を満す．

例えば，整数の集合上で定義された関係 R, P が

$b=a+1$ かつそのときに限り aRb が成立
$b=a+2$ かつそのときに限り aPb が成立

と定義されているとき，

aRc かつ cPb であるような c が存在——$b=a+3$——するときそしてそのときに限り $aRPb$ が成立する．

つぎに，積を用いて，関係 R のべきを定義することができる：

$$R^1=R,\ R^2=RR,\ R^n=R^{n-1}\ R=RR^{n-1}\ (\mathrm{n}>1)$$

R^0 は，**恒等関係** (identity relation) と定義し

$a=b$ が成立ちそのときのみ aR^0b が成立つ．

最後に，関係 R の**推移的閉包** (transitive closure) R^+ は

(2.6.1)　　ある $n(>0)$ に対して aR^nb が成立し，そのときに限り aR^+b が成立する．

で定義する．明らかに aRb ならば aR^+b である．R^+ が実際に推移的であることを示すことは練習問題にしよう．われわれが推移的閉包に対して，R^+ という記号を用いた理由は明らかだろう．関係を順序対の集合と考えると

$$R^+=R^1\cup R^2\cup R^3\cup\cdots$$

である．ここで関係 R の反射的推移的閉包 R^* をつぎのように定義する：

$a=b$ または aR^+b が成立つときそしてそのときに限り aR^*b が成立つ．したがって

$$R^*=R^0\cup R^1\cup R^2\cup\cdots$$

となる．

文法における関係

何故，関係に関心があるのか．第一に記号 $\overset{+}{\Rightarrow}$[(2.3.6) 参照] は，まさに (2.3.5) で定義した関係 $\Rightarrow$ の推移的閉包となっている！　第二に，後で構成しなければならない文法に関連した記号の集合に重要な集合がいくつかある．これらの集合は，きわめて単純な関係とその閉包を用いて，容易に定義することができる．次節で単純な関係から全ての集合を作り出すことができる簡単なアルゴリズムを一つ与えよう．ただこれらの関係は有限集合の上で定義されていなければならない．これを示す前にわれわれが問題にする集合を眺めてみよう．

ある文法とある非終端記号 U が与えられると，U の各導出の先頭の記号の

集合を知る必要がある．したがって，もし $U \doteqdot Sx$ であれば，Sx は U の導出であり，S が先頭の記号の集合の要素である．この集合を *head* (U) と書く：

(2.6.2)　$head(U) = \{S \mid U \doteqdot S\cdots\}$

（ここで"$\cdots$"は関心のない記号列（空の場合もある）を表していることを想起せよ.）大抵の目的に対してこの定義で十分であろう．しかしながらこの定義は語彙 V からできる記号列の無限集合 V^* の上の関係 $\doteqdot$ によっており（もっとも集合 *head* (U) は有限である)，この集合を構成するのは大仕事になるかもしれない．そこでつぎのように有限集合の上の別な関係によって *head* (U) を定義し直そう．与えられた文法の有限な語彙の上の関係 FIRST をつぎのように定義する：

(2.6.3)　規則 $U ::= S\cdots$ が存在しそのときに限り U FIRST S が成立つ.

つぎに推移的閉包の定義によって

(2.6.4)　規則の連鎖 $U ::= S_1\cdots$, $S_1 ::= S_2\cdots$, $\cdots$, $S_n ::= S\cdots$ が存在しそのときに限り U FIRST$^+$ S が成立する.

明らかに $U \doteqdot S\cdots$ が成立ちそのときにかぎり U FIRST$^+$ S が成立することを意味している.

関係 FIRST$^+$ は，これを集合と考えると，語彙 V の全ての記号 U に対して (2.6.2) の集合 *head* (U)——*head* $(U) = \{S \mid (U, S) \in \text{FIRST}^+\}$ ——を完全に定義していることに注目してほしい.

(2.6.5)　**例**　　関係 FIRST と FIRST$^+$ を説明するために，規則をいくつかあげ，その規則の右側に，その規則から導びかれた関係 FIRST を付記する.

	規則	関係
(2.6.6)	$A ::= Af$	A FIRST A
	$A ::= B$	A FIRST B
	$B ::= DdC$	B FIRST D
	$B ::= De$	B FIRST D
	$C ::= e$	C FIRST e
	$D ::= Bf$	D FIRST B

これから FIRST^+ に属するつぎのような記号対が得られる：

$$(A,A)\ (A,B)\ (A,D)\ (B,B)\ (B,D)\ (D,B)\ (D,D)\ (C,e)$$

したがって，A の導出の先頭の記号は A, B, D であり，B に対するそれは，B, D であり，C のそれは e である．■

ここで定義しておきたい集合がまだほかに三つある．これらは全て，FIRST^+ を定義したと同様な方法で定義することができる．第一の集合は，ある記号 U の導出の最後の記号の集合である．これは，全ての記号 U に対して，関係 LAST の推移的閉包である関係 LAST^+ によって定義される．ここで LAST とは

(2.6.7) 規則 $U ::= \cdots S$ が存在しそしてそのときに限り U LAST S が成立つ．

第二の集合は，ある非終端記号 U に対する語に含まれる記号の集合である．これは，関係 WITHIN の推移的閉包である関係 WITHIN^+ によって定義される．ここで WITHIN とは

(2.6.8) 規則 $U ::= \cdots S \cdots$ が存在しそしてそのときに限り U WITHIN S が成立つ．

第三の集合は，ある非終端記号 U の導出となる記号 S の集合である．これは，関係 SYMB の推移的閉包である関係 SYMB^+ によって定義される．ここで SYMB とは

(2.6.9) 規則 $U ::= S$ が存在してそしてそのときに限り U SYMB S が成立つ．

演習問題 2.6

1. 下記の関係は，反射的か．それぞれの関係の推移的閉包および反射的推移的閉包を示せ．

   a) 人間の集合について **より小さい** (IS SMALLER THAN).

   b) 実数の集合について **より大きい** ($>$).

   c) 実数の集合について $a=-b$ のときに限り aRb となるような関係 R.

2. 例 2.3.3 の文法 $G1$ について集合 FIRST⁺, LAST⁺, WITHIN⁺ および SYMB⁺ を構成せよ.

3. 関係における積の操作は結合的であることを証明せよ.

4. 任意の関係 R に対して R^+ は推移的であることを示せ.

5. もし cRb ならば cPb であるような, 任意の推移的関係を P としよう. もし cR^+b ならば cPb であることを証明せよ（したがって R^+ は, R を含む推移的関係の最小のものである).

6. **〜の父である** (FATHER) という関係とそれ自身との 積はどんな関係か.

2.7 関係の推移的閉包の構成

四つの非常に簡単な関係から推移的閉包をつくることによって, 重要な集合を四つ定義した. ここでは, 有限集合の上の関係から推移的閉包を組立てる問題にとりかかろう.

集合 A の上の, ある関係 R とある要素 U について,

$$B = \{S \mid U \ R^+ \ S\}$$

なる集合を構成したい. ここで R^+ は R の推移的閉包である. もし A が無限集合であれば, B 自身は有限であったとしても, これを構成するのは, 非常にむずかしかろう. これは, 集合の2個の要素 a, b に対して, $a \ R^+ \ b$ となるような $a \ R^n \ b$ の n に上限がないからである. しかし, もし A が有限であれば, 必要な鎖の長さに上限をつけることができる. つぎの定理でそれを示す.

(2.7.1) **定理**　A を n 個の記号からなる集合とし, R をその上の任意の関係とする. 2個の記号 S_1 および b に対して, $S_1 \ R^+ \ b$ であれば, $S_1 \ R^k \ b$ となるような正整数 $k \leq n$ が存在する.

証明. $S_1 \ R^+ \ b$ から, $S_1 \ R^p \ b$ となるような整数 $p > 0$ が存在する. すなわち, これは

$$S_1 \ R \ S_2, \ S_2 \ R \ S_3, \ \cdots, \ S_{p-1} \ R \ S_p \text{ かつ } S_p \ R \ b$$

をみたすような記号 $S_2, S_3, \cdots, S_p$ が集合 A の中に存在することを意味している (R^p の定義から). この p の最小値が n より大であると仮定すると,

A には n 個の要素しかないのでこの最小の p の前に二つの整数 i, j $(i < j \leq p)$ が存在して, $S_i = S_j$ とならなければならない. しかし, この場合

$$S_1 \ R \ S_2, \ \cdots, \ S_{i-1} \ R \ S_i,$$
$$S_j \ R \ S_{j+1}, \ \cdots, \ S_{p-1} \ R \ S_p, \ S_p \ R \ b$$

なる関係は $S_1 \ R^k \ b$ $(k = p - (j - i))$ であることを示しており p が最小であるという仮定に矛盾する. したがって $p > n$ という仮定は誤りである. (証明おわり) ■

ブール行列と関係

アルファベットの上の関係を計算機で表現することを考えよう. 最もよい表現法は**ブール行列** (Boolean matrix) B である. これは, 要素 $B[i, j]$ が1または0 (真または偽) の値のみをとる行列である. $n \times n$ ブール行列の "和" とは, 行列を要素ごとに, "論理和" をとることから成り, 行列

$$D = B + C$$

の要素 $D[i, j]$ は

$$D[i, j] := B[i, j] + C[i, j] := \textbf{if } B[i, j] = 1 \textbf{ then } 1 \textbf{ else } C[i, j]^{\dagger}$$

と定義される.

$n \times n$ ブール行列の "積" とは, 算術的な行列積の演算で, 積を "論理積", 和を "論理和" で置換えた演算である. したがって B, C が $n \times n$ ブール行列であり $D = BC$, とすると D の要素 $D[i, j]$ は

$$D[i, j] = B[i, 1] * C[1, j] + B[i, 2] * C[2, j] + \cdots + B[i, n] * C[n, j]$$

で定義される. ここで $*$ は

$$a * b := \textbf{if} \quad a = 0 \quad \textbf{then } 0 \quad \textbf{else } b$$

という演算を示す.

例えば,

$$B = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

ならば

† 訳注 記号 := は, ここでは '左辺の演算は右辺の意味である' の意味で用いている.

$$D_1 = B + C = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \qquad D_2 = BC = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

となる．ここでは，$+$ と $*$ に算術演算とちがう意味を与えていることに注意してほしい．

ブール行列の和は明らかに結合的 $(A + (B + C) = (A + B) + C)$ かつ交換可能 $(A + B = B + A)$ である．一方ブール行列の積は，結合的 $(A(BC) = (AB)C)$ であるにすぎない．二つの演算は分配則を満たす $(A(B + C) = AB + AC)$．

n 個の記号 $S_1, S_2, \cdots, S_n$ からなる集合 S 上で定義された関係 R を考えよう．$n \times n$ ブール行列 B を，$S_i\ R\ S_j$ が成り立つときに限り，$B[i, j] = 1$ となるように作る．例えば，(2.6.6) の集合 $\{A, B, C, D, e, d, f\}$ の上で定義された関係 FIRST に対するブール行列 B は，図 2.9 で与えられる．

(2.7.2)　　関係 R の逆は，R を表すブール行列の転置行列によって与えられる．

この表明は明らかである．

つぎに，アルファベット S 上で定義される関係 P, Q を表す行列 B, C の積 $D = BC$ について議論しよう．もし $D[i, j] = 1$ ならば，$D[i, j]$ の定義から，

$$B[i, k] = 1 \quad \text{かつ} \quad C[k, j] = 1$$

となる k が存在することを示している．すなわち $S_i\ P\ S_k$, $S_k\ Q\ S_j$ が成立ち，記号の対 $(S_i,\ S_j)$ が二つの関係 P, Q の積 PQ に属していることを意味している．したがって，つぎの定理が証明された．

	$S_1 = A$	$S_2 = B$	$S_3 = C$	$S_4 = D$	$S_5 = e$	$S_6 = d$	$S_7 = f$
$S_1 = A$	1	1	0	0	0	0	0
$S_2 = B$	0	0	0	1	0	0	0
$S_2 = C$	0	0	0	0	1	0	0
$S_4 = D$	0	1	0	0	0	0	0
$S_5 = e$	0	0	0	0	0	0	0
$S_6 = d$	0	0	0	0	0	0	0
$S_7 = f$	0	0	0	0	0	0	0

図 2.9　関係 FIRST に対するブール行列

$$B = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{matrix} B^2 = \\ B^4 = \\ B^6 = \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{matrix} B^3 = \\ B^5 = \\ B^7 = \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad B^+ = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

図2.10 B から B^+ の計算

(2.7.3) **定理** 同じアルファベットの上で定義された二つの関係の積は，これらの関係を表すブール行列の積によって与えられる．■

一般に $n>1$ のとき R^n は，RR^{n-1} と再帰的に定義されるので，n に関する帰納法によって，行列 $B^n = B\ B\ B\cdots B$ (n 個の積) は，関係 R^n を表している．R^+ と，定理 2.7.1 および 2.7.3 から，つぎの定理が直ちに導かれる：

(2.7.4) **定理** n 個の記号をもつアルファベット S 上の関係 R を表す $n \times n$ ブール行列を B とすると，

$$B^+ = B + BB + BBB + \cdots + B^n$$

で定義される行列 B^+ は R の推移的閉包 R^+ を表す．■

例として，図2.10 に図2.9 の行列 B (文法 (2.6.6) の関係 FIRST) に対する $B, B^2, \cdots, B^7$ と B^+ を示した．B^+ から，FIRST^+ に属するつぎのような記号の対が再び，得られる．

$$(A,A)\quad (A,B)\quad (A,D)\quad (B,B)\quad (B,D)\quad (D,B)\quad (D,D)\quad (C,e)$$

関係とブール行列を用いて，いくつかの相異なる重要な集合を作り上げるアルゴリズムをただ一つ手にすることができた．これらの集合は，後に，構文解析アルゴリズムを構成するために使うことになる．ここで用いられた諸概念は意外なほど単純で，理解が容易である．

演習問題　2.7

1. Warshall(62) は，行列Bに対して $B^+ = B + BB + \cdots + B^n$ を計算する能率のよいアルゴリズムをつぎのように与えた．このアルゴリズムが正しく働らくことを証明せよ．

   1. 新しい行列 A に B を代入せよ．
   2. $i := 1$ とする．
   3. 全ての j に対して，もし $A[j, i] = 1$ ならば，$k = 1, \cdots, n$ に対して $A[j, k] := A[j, k] + A[i, k]$ とする．
   4. $i := i + 1$.
   5. $i \leq n$ ならば，ステップ 3. へ，さもなければ終了．

2.8　文法に対する実際的な制限

この節では，文法になんらかの実際的な制限を置くことを議論しよう．このような制限をつけても実際には，文法によって記述される言語の集合に制限が及ばない．このような条件について，文法を検査するアルゴリズムを概観する．またこの節では，構文解析の問題に入る前に，読者に文法の取扱いにもっと慣れてもらうようにしたいと考えている．

文法の中で $U ::= U$ のような規則は明らかに不要なものである．それだけでなく，この規則は，文法をあいまいにする．したがって，以後本書ではつぎの仮定を置く：

(2.8.1)　文法は規則　$U ::= U$ を含まない．

文法には，余分な規則——すなわち，文を導出する過程で全く用いられないような規則——が含まれていてもよい．例えば，下記の文法 $G4[Z]$ において，非終端記号 $\langle d \rangle$ は，それがどの規則の右辺にも現れないので，文を導出するのに用いることがない．

$$
\begin{aligned}
Z \ &::= \langle b \rangle\ e \\
\langle a \rangle &::= \langle a \rangle\ e \mid e \\
\langle b \rangle &::= \langle c \rangle\ e \mid \langle a \rangle\ f \\
\langle c \rangle &::= \langle c \rangle\ f \\
\langle d \rangle &::= f
\end{aligned}
$$

文法の文を構文解析することは，その文法が余分な規則を持っていなければ

ば，ずっと容易である．余分な規則を持った，プログラム用言語の文法は，なにか誤りを含んでいるに違いないと主張できるのも道理がある．したがって，文法の中に余分な規則の有無を調べるアルゴリズムは，言語の設計者にとって有用である．

非終端記号 U が，ある文の導出中のどこかに現れるためには二つの条件を満たさねばならない．第一に，いずれかの文形式の中に U が現れなければならない．すなわち

(2.8.2)　　x, y を適当な記号列として $Z \overset{*}{\Rightarrow} xUy.$

ここで Z はこの文法の出発記号である．第二に，終端記号のみからなる記号列を，U から導出することができなければならない．すなわち

(2.8.3)　　$V_T{}^+$ に属する t が存在して，$U \overset{*}{\Rightarrow} t.$

もし，非終端記号がこれら2条件のいずれかを満たさなければ，左辺に U を含むような規則は，どんな導出にも使用することができないのは明白である．一方全ての非終端記号が，これらの条件を満たすならば，この文法は余分な規則を含まない．なんとなれば，もし $U ::= u$ を任意の規則とすると，まず，$Z \overset{*}{\Rightarrow} xUy \Rightarrow xuy$ である．つぎに，この記号列 xuy に含まれている全ての非終端記号から，終端記号だけで構成される記号列を導出できるから，

$$xuy \overset{*}{\Rightarrow} t$$

となる記号列 t が $V_T{}^+$ の中に存在する．これらをまとめて $Z \overset{+}{\Rightarrow} t$ となる．この導出には，規則 $U ::= u$ が使われていた．

上の文法 $G4$ で，$\langle c\rangle$ は条件 (2.8.3) を満たさない．それは，もし $\langle c\rangle$ が直接導出で置換えられるとすれば，$\langle c\rangle f$ とならなければならず，これが，終端記号列に到達しないのは明らかである．とにかくどんな文を生成するのにも使われないような規則を全て取除くならば，つぎのような規則が残されるだろう：

$$Z ::= \langle b\rangle e,\ \langle a\rangle ::= \langle a\rangle\, e,\ \langle a\rangle ::= e,\ \langle b\rangle ::= \langle a\rangle f$$

(2.8.4)　**定義**　　文法 G において，全ての非終端記号がそれぞれ (2.8.2) および (2.8.3) を満たすならば，G は**既約** (reduced) であるという．■

本書では，以後に取扱う文法は全て既約であるものと仮定する．

関係 WITHIN を (2.6.8) のように定義すると Z WITHIN$^+$ U のときそ

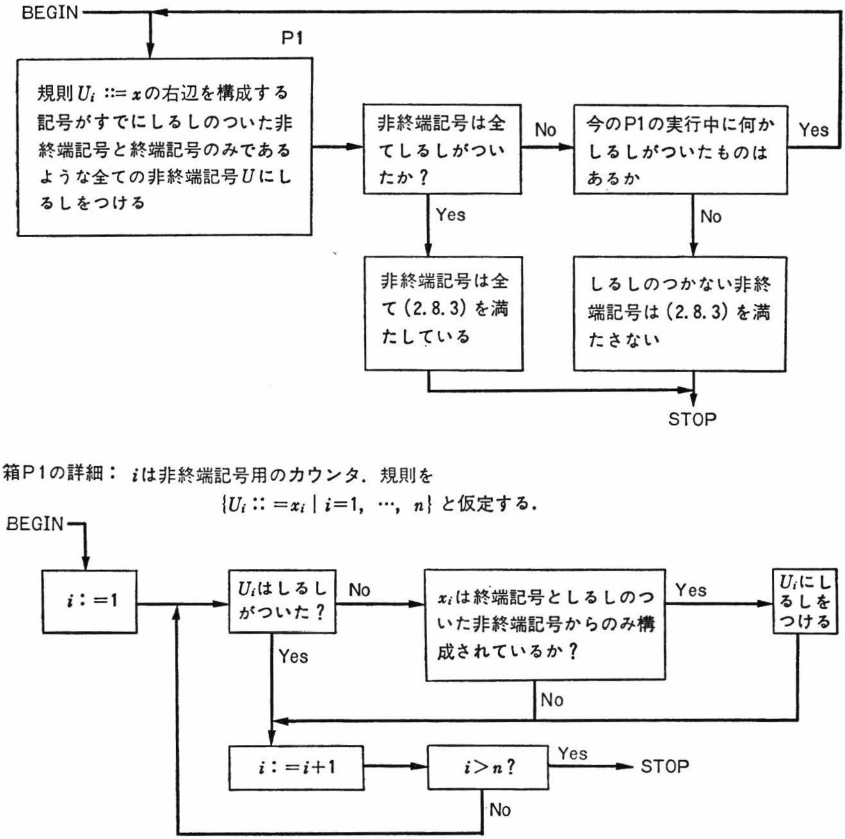


図2.11　(2.8.3) を調べるアルゴリズム

してそのときに限り非終端記号 U は条件 (2.8.2) を満たすことに注意しよう．したがって，ある文法で条件 (2.8.2) を検査するには，前節の結果を使うことができる．

図2.11 は，一連の規則にある非終端記号について，条件 (2.8.3) を検査するアルゴリズムの流れ図である．これは，まず，V_T^+ に属する t に対して U ::= t という規則が存在するような非終端記号 U を全て，何らかの方法で“しるしをつける”ことから始める（箱 P1 の最初の実行）．このような非終端記号は明らかに (2.8.3) を満たしている．つぎに，まだしるしのついていない非終端記号について，x がすべて終端記号と，すでにしるしのついた非終端

記号だけからなる記号列であるような，規則 $U ::= x$ があるかどうか調べる（箱 P1 の 2 回目の実行）．このような非終端記号は明らかに (2.8.3) を満たしている．この過程を，全ての非終端記号にしるしがつくか，箱 P1 を実行して，1 個も非終端記号にしるしをつけられなくなるまで繰返す．最後までしるしのつかなかった非終端記号は，条件 (2.8.3) を満たさない．

演習問題 2.8

1. 下記の文法を考えよう．

$$
\begin{aligned}
Z &::= E + T \\
E &::= E \mid S + F \mid T \\
F &::= F \mid FP \mid P \\
P &::= G \\
G &::= G \mid GG \mid F \\
T &::= T * i \mid i \\
Q &::= E \mid E + F \mid T \mid S \\
S &::= i
\end{aligned}
$$

この文法につぎのような操作を施せ．

A. a) 関係 WITHIN$^+$ を計算せよ．

b) (2.8.2) を満たさないような非終端記号を含む（左辺，右辺いづれも）規則を全て取除け．

B. a) 図 2.11 のアルゴリズムを実行せよ．

b) (2.8.3) を満たさないような非終端記号を含んでいる規則を全て取除け．

c) ステップ **B.** の以前には非終端記号で，ここでは終端記号になった記号を並べよ．

d) ステップ c) で列挙した記号を含んでいる規則を全て取除け．

e) ステップ c) および d) を，規則が除去できなくなるまで繰返せ．

C. ステップ **A.** および **B.** を，除去できる規則がなくなるまで繰返せ．

D. ステップ **C.** で得られた文法を変形して (2.8.1) を満たすようにせよ．

この文法の言語を書下せ．

2.9　その他の構文表記法

さまざまな言語の構文を記述するために，このほかいくつかの表記法が用いられている．これらをここで簡単に概観してみたい．これらの表記法は，BNF と能力が同じである――すなわちこれらの表記法で表現できることは BNF でも表現できる．種々の点で，これらは，純粋な BNF よりも読み易く，本書の中で，しばしば，これらを使うことになろう．

中かっこ

BNF では，任意個の要素の並びを示すために再帰的定義を行う．すなわちつぎのような書き方をする．

(2.9.1)　〈名前の並び〉 ::= 〈名前〉|〈名前の並び〉, 〈名前〉

(2.9.2)　〈整数〉　　　::= 〈数字〉|〈整数〉〈数字〉

記号列 x が任意の回数現れることを意味するために x を囲む超言語記号として中かっこ “{” および “}” を採用すると，(2.9.1) と (2.9.2) はつぎのように書直すことができる．

(2.9.3)　〈名前の並び〉 ::= 〈名前〉{, 〈名前〉}
　　　　〈整数〉　　　::= 〈数字〉{〈数字〉}

二様の表記法は，構造的に等価なものであると仮定する．ときには，閉じかっこの後に下つき添字および（または）上つき添字をつけて，かっこの中の記号列の繰返し回数の最小値と最大値を示すことがある．例えば，FORTRAN の名前† は，最大6文字の英数字からできているが，これをつぎのように記述することができる：

(2.9.4)　〈FORTRAN 英字名〉 ::= 〈英字〉{〈英数字〉}$_0^5$
　　　　〈英数字〉　　　　　::= 〈英字〉|〈数字〉

一方これを BNF で表現しようとすると，見苦しいものになる：

〈FORTRAN 英字名〉 ::= 〈英字〉|〈英字〉〈英数字の並び〉
〈英数字の並び〉　　::= 〈英数字〉|〈英数字〉〈英数字〉

† **訳注**　FORTRAN の JIS では，**名前**（identifier）の代りに**英字名**（symbolic name）と名付けている．

|〈英数字〉〈英数字〉〈英数字〉
|〈英数字〉〈英数字〉〈英数字〉〈英数字〉
|〈英数字〉〈英数字〉〈英数字〉〈英数字〉〈英数字〉

上付き添字や下付き添字をやめて1次元的な表現にするために，中かっこの中の記号列の最小繰返し回数は0と約束し，最大回数の方は閉じかっこのすぐ後ろに空白ぬきで書くことにすると，(2.9.4) はつぎのように書ける：

〈FORTRAN 英字名〉::= 〈英字〉{〈英数字)}5

これは

〈FORTRAN 英字名〉::= 〈英字〉{〈英数字〉}　5

とは異なっていることに注意．つまり後者では閉じかっこと5の間に空白が置かれている．

中かっこを使うと超言語記号 | を中かっこの内部に用いて，選択を示すことができるようになる．$\{x|y|z\}$ と書くと，これは，x, y, z のいずれかの記号列が任意個並んだ記号列をさらに並べた記号列を意味する．この表記法を用いると FORTRAN の英字名の構文はつぎのように定義し直すことができる：

〈FORTRAN 英字名〉::= 〈英字〉{〈英字〉|〈数字〉}5

大かっこ

大かっこ“[”と“]”は，文字列を任意に付加えることを示すために用いられる．したがって中かっこの直後に1をつける表記法と等価である．ここで ALGOL 型の簡単な繰返し文——初期値や増分値を省略することができる——の構文はつぎのように書けるだろう．

〈繰返し文〉::= FOR〈変数〉[:=〈算術式〉][STEP〈算術式〉]
UNTIL〈算術式〉DO〈実行文〉

超言語記号としての小かっこ

生成規則の右辺では，連接の操作の方が選択の操作より優先する．$AB \mid C$ は AB または C を採ることを意味する．必要があれば，小かっこを用いて，ちょうど普通の算術式で行っているように，通常の優先順位を変更させることができる．したがって，$A\ (\ B \mid C\ )$ は AB または AC を意味する．超言

語記号としての小かっこは，右辺で，共通要素を括り出すのに役立つだろう．これをしばらく調べてみよう．しかしながら，現に定義しようとしている言語の終端記号として小かっこを使いたいときには，問題が生じる．特に注意しないかぎり，小かっこは，普通の終端記号して使うことにしよう．

共通要素の括り出し

もし，小かっこを超言語記号として使うならば，

$$U ::= xy \ | \ xw \ | \ \cdots \ | \ xz$$

のような規則を

$$U ::= x \ (y \ | \ w \ | \ \cdots \ | \ z)$$

と書くことができる．ここでは，全ての選択枝に共通な頭部 x を括り出した．小かっこは，算術式の場合と同様に，どんな水準までも，入れ子にすることができる．例えば，$y = y_1y_2$，$w = y_1y_3$ とすると，上の規則はつぎのように書換えることができよう

$$U ::= x \ (y_1 \ (y_2 \ | \ y_3) \ | \ \cdots \ | \ z).$$

後で見るように，このような共通要素の括り出しによって，構文解析が容易になることがしばしばである．

もし，E ::= T|E + T という規則は，少くとも1個の T と，その後に任意個の + T が続くことを意味していると考えるならば，この規則を

E ::= T { + T }

と書直すことができる．さらに，E ::= T|E + T|E − T は

E ::= T { (+ | −) T }

と書直すことができる．このような変換は，規則の中から左再帰性を取除くので，下向き構文解析では有用である．

終端記号としての超言語記号

::= とか | とか { などのような超言語記号が同時に終端記号に含まれるような言語を記述しなければならないことがしばしばある．このような場合に

は，その記号の前後を引用符で区切り，超言語記号と区別する．例えば BNF の規則の一部はつぎのように記述できる．

〈規則〉::= 〈左辺〉′::=′〈右辺〉

ここではもちろん，引用符を超言語記号としている．これ自身を引用符ではさんで終端記号として表現しなければならないときは，引用符を2個並べる．したがって〈引用符〉::= ′′′′ という規則は，非終端記号〈引用符〉を記述している．この規則から導びかれる記号列は，ただ1個の引用符 ′ からなる記号列だけである．

2.10 形式言語理論の概観と参考文献

形式言語理論は Chomsky (56) の最初の出版以来急速に発展した．発表された論文は大部分，プログラム用言語によるプログラムを構文解析するというわれわれの問題には直接関係はない．これらの論文はむしろ，言語や文法の数学的理論を取扱っている．いろいろな性質を持った言語のクラスが定義され，このような言語は，そのクラスの言語だけを生成する文法によって特徴づけられ，そのクラスの言語だけを認識するオートマトンによって特徴づけられている．典型的な問題としては，L_1 と L_2 が，あるクラス T の二つの言語であるならば，両者の合併（あるいは共通部分）も T に属するか？ 文法はあいまいでないか？ クラス T の任意の文法について，その文法があいまいか否かを知るアルゴリズムを与えることができるか？ 二つの文法が，何らかの意味で等価か？ などがある．Hopcroft と Ullman (69) は，この主題に関する良い教科書である．Ginsburg (66) には，より多くの内容が盛り込まれているが，前者より簡潔で，読みこなすのはよりむずかしい．この方面の論文をたくさん載せた雑誌には，*Information and Control* と *Journal of the ACM* がある．†

Chomsky (56) は，文法によって，言語の4通りのクラスを定義した．これはつぎのような 4-組 (V, T, P, Z) である．

1. V はアルファベットである．
2. T は V に含まれ，終端記号のみからなるアルファベットである．
3. P は有限個の書換え規則の集合である．

---

† **訳注** そのほか *Mathematical System Theory, Acta Informatica, Journal of Computer and System Science, Computer and Information Science* など数多くある．

4.　Z は出発記号であり，差集合 $V-T$ に属する．

ある文法の言語とは，Z から生成される終端記号からなる記号列の集合である．文法の四つの型の違いは，P で許される書換え規則の形式にある．第一に P に含まれる規則が

(2.10.1)　　$u ::= v$ ただし u は V^+ に属し，v は V^* に属する

の形式をしたものであれば，文法 G は **0型** (type 0) **文法**あるいは**句構造文法** (phrase structure grammar) という．すなわち，左辺 u も記号の系列であってよく，右辺は空であってよい．

句構造文法を扱った仕事は比較的少ない．書換え規則をつぎの形式

(2.10.2)　　$xUy ::= xuy$，ここで U は $V-T$，x，y は V^*，u は V^+ にそれぞれ属する．

に限定すると，さらに興味深い **1型** (type 1) **文法**，あるいは，**文脈依存文法** (context sensitive grammar, context dependent grammar) が得られる．文脈に依存するということばは，$x \cdots y$ という文脈のときにのみ U を u に書換えることが許されるという事実を示している．さらに制限を加えてゆくと，われわれが使っているものによく似た文法のクラスができる．全ての規則がつぎの形式

(2.10.3)　　$U ::= u$　ここで U は $V-T$，u は V^* にそれぞれ属する

であれば，文法は**文脈自由** (context free) と呼ばれる．このクラスが文脈自由と呼ばれる理由は，U がその前後の文脈に無関係に u に書換え可能であるからである．文脈自由文法には，$U ::= \varepsilon$ (ε は空記号列) という形式の規則があってよい．われわれの文法では，これは許さない．それは，これによって，術語や証明が必要以上に複雑になるからである．文脈自由文法 G が与えられると $L(G1) = L(G) - \{\varepsilon\}$ によって，**ε なしの文法** (ε-free grammar) $G1$ (これはわれわれの文法の1種) を構成することができる．さらにもし G があいまいでなければ，$G1$ もあいまいでなく，実際上制限ではない．

さらにつぎのように規則を制限すると

(2.10.4)　　$U ::= N$ あるいは $U ::= WN$，ここで N は T に属し U と W は $V-T$ に属する．

3型 (type 3) **文法**，あるいは**正規文法** (regular grammar) が得られる．

正規文法は，言語理論とオートマトン理論の両方で，基本的な役割を演じている．正規文法によって生成された記号列の集合は，**有限状態オートマトン**(finite state automaton) と呼ばれる機械によっても受理されるし，逆もいえる（有限状態オートマトンは次章で定義する）．したがって，われわれは，オートマトンによって，このクラスの文法を特徴づける．正規文法によって生成された正規言語† は**正規集合**† とも呼ばれる．もし L_1 と L_2 が正規集合であれば，$L_1 \cup L_2$，$L_1 \cap L_2$，$L_1 - L_2$，$L_1 \cdot L_2 = \{xy \mid x \in L_1$ かつ $y \in L_2\}$，さらに $L_1{}^* = \{\varepsilon\} \cup L_1 \cup L_1{}^2 \cup L_1{}^3 \cup \cdots$ も正規集合であることが知られている．

ここで定義した文法の4通りのクラスは，段階的に制限を加えていって得られた．すなわち，文脈に依存しない句構造言語があり，文脈自由でない文脈依存言語があり，正規でない文脈自由言語がある．形式言語理論でなされてきた研究の大部分は，文脈自由言語と正規言語，あるいは，その部分集合を扱っており，われわれもこれらに限ることにしよう．

形式言語理論における基本的な問題の一つは文法があいまいであるか否かである．不幸にして（立場によっては，幸いにも）この問題は，文脈自由文法に対しては，**決定不能である** (undecidable) ことが証明されている．これは，任意の文法が与えられたときに，それがあいまいであるかどうかをつねに有限の手続きで決定する効果的なアルゴリズムを与える（プログラムを書く）ことができないことを意味している．文脈自由文法に関する興味ある問題の多く（二つの言語が同等であるか，二つの言語が共通部分を持たないかなど）は決定不能であることがわかっている．こうなったときには，問題となっている性質が決定可能であるような，言語の部分クラスを探すことになる．形式言語理論における決定不能性の証明は直接であれ間接であれ大部分 Post (46) の唯一の定理によっている．

任意の文脈自由文法があいまいであるかどうかを決定する問題は決定不能であることが証明されている．このつぎの問題は，任意の文脈自由言語に対してあいまいでない文法が存在するかどうかであろう．あいまいでない文法が存在しないような言語があるということは，最初に Parikh (61) によって示された．このような言語を**本質的にあいまいである** (inherently ambiguous) と称する．例えば

---

† **訳注**　正規言語，正規文法，正規集合，正規表現は主に工学畑で用いられることが多いようである．岩波数学辞典では，regular に対する日本語は正則となっている (例．正則関数)．したがって同じことを正則言語，正則文法，正則集合，などと表した本も多いので注意が必要である．

$$\{ a^i b^i c^j \mid i, j \geq 1 \} \cup \{ a^i b^j c^j \mid i, j \geq 1 \}$$

がその例である．

言語の文を構文解析する当面の問題を扱った文献はたくさんある．この問題は，後につづく数章で議論しよう．

2.11 復　　習

この章では以後の5章のための基礎的な概念を説明した．読者は先に進む前に本章で与えた素材をよく理解しておいてほしい．これらの概念を復習する助けにするために，下記に，その定義が出現した順にページをつけて，一覧表を作っておいた．

超言語 (metalanguage)	13
アルファベット (alphabet)	16
記号 (symbol)	16
記号列 (string)	16
空記号列 (empty string)	16
記号列の長さ (length of a string)	16
連接 (catenation)	17
記号列の頭部，後部 (head, tail of a string)	17
真の頭部，真の後部 (proper head, tail)	17
二つの集合の積 (product of two sets)	17
記号列のべき (power of a string)	17
集合の閉包 (closure of a set)	18
正の閉包 (positive closure)	18
生成規則 (production)	19
規則 (rule)	19
規則の左辺 (left part of a rule)	19
規則の右辺 (right part of a rule)	19
文法 (grammar)	19
出発記号 (distinguished symbol)	20
語彙 (vocabulary)	20
非終端記号 (nonterminal)	20

終端記号 (terminal)　20
BNF (Backus-Normal Form, Backus-Naur Form)　20
直接導出 $\Rightarrow$ (direct derivation)　21
導出 $\overset{+}{\Rightarrow}$ (derivation)　21
導出の長さ (length of a derivation)　21
文形式 (sentential form)　22
文 (sentence)　22
言語 (lauguage)　22
句 (phrase)　23
単純句 (simple phrase)　23
把手 (handle)　24
再帰的文法 (recursive grammar)　24
再帰的な規則 (recursive rule)　24
左 (右) 再帰的な規則 (left (right) recursive rule)　24
構文の木 (syntex tree)　25
枝, 節点 (branch, node)　25
端点 (end node)　25
端枝 (end branch)　26
節点の兄弟 (brother of a node)　26
節点の息子 (son of a node)　26
節点の父 (father of a node)　26
部分木 (subtree)　26
あいまいな文 (ambiguous sentence)　29
あいまいな文法 (ambiguous grammar)　29
本質的にあいまい (inherently ambiguous)　29
文形式の構文解析 (parse of a sentential form)　30
下向き構文解析 (top-down parse)　32
上向き構文解析 (bottom-up parse)　32
正準構文解析 (canonical parse)　33
関係 (relation)　35
逆関係 (transpose of a relation)　36
反射的関係 (reflexive relation)　36
推移的関係 (transitive relation)　36

第 3 章

走　　査　　部

3.1 序　　論

走査部とは，コンパイラの一部で，原始プログラムを文字の系列として読込んで，原始プログラム記号（名前，予約語，整数，＊や＋，＊＊や/＊のような区切り記号）に組上げる働きをするものである．**記号** (symbol) の代りに**しるし** (token) とか **原子記号** (atom) などの用語を使うこともある．走査部の純粋な形態は，原始プログラムについて簡単な**字句解析** (lexical analysis)を行うところにあり，これは構文解析と対比されるものである．したがって，字句解析部 (lexical analyzer) と呼ばれることもある．

当然のことであるが，なぜ字句解析が構文解析の中に組込まれていないかとたずねる人がいるかもしれない．結局のところ，記号の構文を記述するのに BNF を用いることができる．例えば，FORTRAN の英字名はつぎのように記述できる．

(3.1.1)　〈英字名〉::=〈英字〉{〈英字〉|〈数字〉}5

字句解析を構文解析から分離している納得の行く理由はいくつかあげることができる．

1. コンパイル時のかなりの部分は，文字を走査するのに費される．二つの解析を分離すれば，この時間を減少させることにのみ集中することができるようになる．例えば，一案として，走査部の一部または全部をアセンブリ言語でプログラムすることもある．これも，分離がなされている方が容易である（例えば，IBM 360 では，TRT 命令1個で，1～256文字を並べて，空白でない文字を探すことができる）．もちろん，そのコンパイラが，実際にあまり使われる見込みがないのならば，アセンブリ言語を使う

ことは奨めない.

2. 記号の構文は，非常に簡単な文法で記述できる．もし走査と構文の認識とを分離すれば，特にこの文法によく合った能率のよい構文解析技法を展開でき，さらにこのような能率のよい技法を用いる走査部を自動的に組立てる方法を開発できる.

3. 走査部への入力は文字列で，出力は記号なので，構文解析部は，実際にその各段で何をなすべきかについてより多くの情報が得られる．さらに記号を構文解析するのに必要な文脈の検査は，形式的な構文解析部が行うよりも，特別な方法で行う方が容易な場合がある．例えば，FORTRAN の実行文 DO10I = ··· の = 記号の後に ，と (のどちらが先に現れるかを調べればこれの意味するところを認識するのはたやすい.

4. 高水準言語が発達してくると，字句の性質と構文の性質の両方に念を入れなければならなくなってくる．両者を分離しておくと，これらを独立に究明することができる.

5. 同じ言語に対して，二つの別々の機械上の表現（金物表現）があることがしばしばある．例えば，ALGOL を作成するとき，予約語は引用符に囲まれ，空白は本質的に意味を持たず，完全に無視されることがある．場合によっては，予約語は名前として用いることができず，予約語同士あるいは名前との間に少くとも 1 個以上の空白を置かねばならないこともある．紙テープやカードやオンラインのタイプライタ上の金物表現はそれぞれ全く異なっている可能性が大きい.

　二つの処理を分離しておけば，一つの構文解析部に対して，複数の走査部を書くことが可能になる（走査部の方が構造が簡単で書くのが易しい）ので，原始プログラムの表現や入力装置の別に一つづつ走査部を書けばよい．それぞれの走査部はそれぞれの金物表現の記号を，構文解析部で用いる共通の内部形式に変換する.

走査部は原始プログラムの字句解析だけを行う独立したパスとしてプログラムしてもよい．構文解析部にはその結果である内部記号の形式に変換された原始プログラムが入った表を渡すことになる．あるいは，構文解析部が新しい記号を要求する時いつでも呼出すサブルーチン SCAN としてもよい（図 3.1）. SCAN は，呼出されると，つぎの原始プログラムの記号を認識してこれを構文解析部に返す．こちらの方法の方が一般的には都合がよく，それは，原始プ

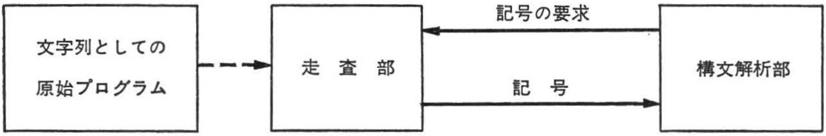


図3.1 構文解析部のサブルーチンとしての走査部

ログラムの内部形式の全体が組立てられて，記憶装置の中にたくわえられている必要がないからである．以後この章では，走査部は，後者のように作成されるものと仮定しよう．

この章を通じて，原始プログラムは，欄にこだわらない自由な書式で書かれ，連続した文字の流れであると仮定しよう．欄を固定した書式や，カードの終端が文の切れ目を意味するような場合（例．FORTRAN）の修正や追加については，3.3節の最後で取扱う．記号とそれより上位概念である構成要素との間のちがいが不明確なことがしばしばある．例えば，整数と〈整数〉.〈整数〉の形式をした実数を両方とも記号とみなしたり，整数は記号であるが，実数は，上位の構成要素とみなしたりする．主に表現を簡単にするため，このような場合を適宜使い分けることにしたい．

正規文法の使用

プログラム用言語の大部分の記号は，つぎのいずれかに分類される：

名前
予約語（これは名前の部分集合である）
整数
1文字の区切り記号（+，−，(，)，/，など）
2文字の区切り記号（//，/*，**，:=，など）

これらの記号は，つぎのような簡単な規則で記述することができる：

〈名前〉 ::= 英字 | 〈名前〉 英字 | 〈名前〉 数字
〈整数〉 ::= 数字 | 〈整数〉 数字
〈区切り記号〉 ::= + | − | (|) | / | ･･･
〈区切り記号〉 ::= 〈SLASH〉/ | 〈SLASH〉* | 〈AST〉* | 〈COLON〉=
〈SLASH〉 ::= /
〈AST〉 ::= *
〈COLON〉 ::= :

もちろん，これらの規則はもっと簡単にしようと思えばできるが，ここでは全ての規則が，

$$U ::= T \quad \text{または} \quad U ::= V\ T$$

の形式となるように書いた．ここで T は終端記号で V は非終端記号である．2.10 節を思い出してもらえば，このような規則を持った文法は，**3型** または**正規**文法と呼ばれている．大抵のプログラム用言語の構文は，このような形式で明記することができて，正規文法の文を構文解析する能率のよい方法を見出すのに都合がよい．では，能率のよい構文解析の機構の設計を始めよう．

3.2 正規表現と有限状態オートマトン

この節には形式的な証明を与えていない結果がいくつか含まれている．ここでの主な関心は，走査部をプログラムすることにあり，実際の証明は必要ない．詳しい証明が必要な場合は，3.6 節の文献を参照されたい．

状 態 図

つぎのような正規文法 $G[Z]$ を考えよう：

$$
\begin{aligned}
Z &::= U\ 0 \mid V\ 1 \\
U &::= Z\ 1 \mid 1 \\
V &::= Z\ 0 \mid 0
\end{aligned}
$$

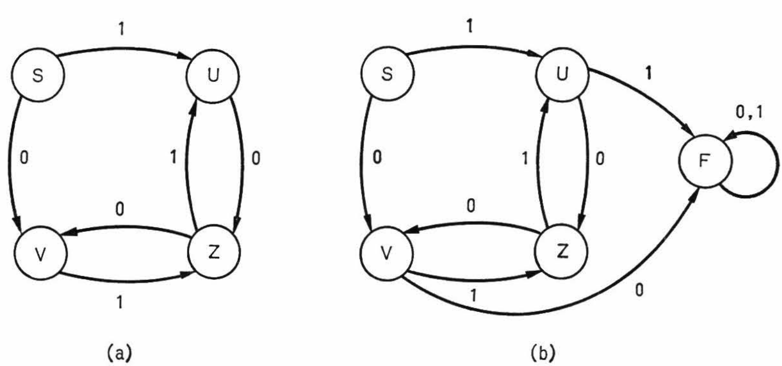


図3.2 状 態 図

ちょっとあたってみれば，この言語は 01 または 10 の対を並べたものであることがわかる．すなわち

$$L(G)=\{\,B^n \mid n>0\,\}\quad ただし\quad B=\{\,01,10\,\}$$

G の文を認識するのに図3.2a のような**状態図** (state diagram) を描くとよい．この図で G の非終端記号は，それぞれ，節点または**状態** (state) で表されている．このほか，**初期状態** (start state) S がある（文法には，S という非終端記号はないものと仮定する）．G のそれぞれの規則 $Q ::= T$ に対して初期状態 S から状態 Q に向う T と名札をつけた矢印が存在する．また G の，それぞれの規則 $Q ::= RT$ に対して，状態 R から状態 Q に向かう T と添記した矢印が存在する．

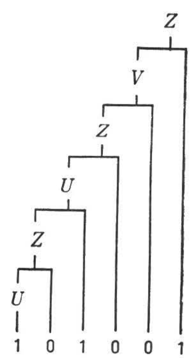
われわれは，つぎのようにして，記号列 x を構文解析，あるいは認識するために状態図を用いる：

1. 初期状態 S から始める．これが最初の“現在の”状態である．x の最も左側の文字から始めて，x の右端に至るまで，つぎのステップを繰返す．
2. x のつぎの文字を走査する．その文字を名札として付けた矢印に沿って現在の状態からつぎの状態に進む．

もし，ステップ2. を繰返すうちに，そのような矢印がなくなれば，x は文ではなく，処理を停止する．もし，x の右端に達したならば，最後の状態が Z でありそのときに限り x は文である．

ステップ	現在の状態	x の残り
1	S	1 0 1 0 0 1
2	U	0 1 0 0 1
3	Z	1 0 0 1
4	U	0 0 1
5	Z	0 1
6	V	1
7	Z	

(a) 構文解析



(b) 構文の木

図3.3 101001 の構文解析と構文の木

読者は，これが，上向き構文解析と認めるだろう．第1ステップを除いた各ステップにおける把手は，現在の状態名につぎに入ろうとする記号が続いたものである．これから還元される記号が，つぎの状態名である．例えば，101001という文を構文解析してみよう．図3.3**a**の各行は，ステップ2に入る時点の構文解析の状態を示している．

この構文解析は，規則が単純なので，簡単である．非終端記号は，右辺の最初の記号としてしか現れないので，最初のステップはいつでも，文の最初の文字を非終端記号に還元する．以後つづくステップでは $V ::= UT$ という規則を用いて，文形式 UTt の最初の二つの記号を非終端記号 V に還元する．この還元を行う時の現在の状態名は U であり，つぎの状態名は V である．各規則の右辺は一意的なので，還元されるべき記号は一意的である．正規文法の文に対する構文の木は，つねに，図3.3**b**に与えた木のような形をしている．

各段で，正しい名札のついた矢印があるかどうか調べる手数を省くためにさらにもう一つ F と名付けた状態（FAILURE 失敗の頭文字）を設けて，全ての状態から F に向かう必要な矢印を任意に加えることができる．F から F に向かい，全ての文字を名札としてつけた矢印も付加える．これによって図3.2a は，図3.2**b**のように変る．

有限オートマトンの決定

状態図の取扱いを容易にするために，この概念をもう少し形式化してみたい：すなわち，状態，入力文字，初期状態 S，現在の状態 Q と入力文字 T が与えられたとき次の状態を示す写像 M，そして上例の Z に対応する最終状態などである．

(3.2.1) **定義**　（決定性）**有限状態オートマトン（fa）**とは，5-組（K, V_T, M, S, Z）である．ここで，それぞれの記号はつぎのような意味をもつ：

1. K は**状態**（state）と呼ばれる要素からなるアルファベットである．
2. V_T は，**入力アルファベット**（input alphabet）と呼ばれるアルファベットである（記号列あるいは，文に現れることのできる文字の集合）．
3. M は $K \times V_T$ から K への写像（mapping）である（$M(Q, T) = R$ とは，状態 Q に対して入力文字が T であれば状態が R に切換ることをいう）．

4. S は，**初期状態** (start state) である (S は K に属していなければならない).
5. Z は最終 (final) 状態の空でない集合で，これらの状態は全て K に属していなければならない. ■

さらに，入力記号列 t に対して fa (あるいは状態図) を動かすことの意味を形式的に定義することもできる. これは，入力文字に応じて状態の切換えを行う方法を示す写像の概念を拡張することによって定義する. すなわち

全ての状態 Q に対して　$M(Q, \varepsilon) = Q$

$V_T{}^*$ に属する t と V_T に属する T のそれぞれに対して

$$M(Q, Tt) = M(M(Q, T), t)$$

前者は，入力文字が空であれば，同じ状態に留まることを示している. 後者は，状態 Q で，入力文字列 Tt がであれば，まず写像 M を，状態 Q と，入力記号列 T に適用して，状態 P を得，つぎに P と入力記号列 t に M を適用して状態 $M(P, t)$ を得ることを示している. 最後に，P が最終状態の集合 Z に属する状態であるとき，もし $M(S, t) = P$ であれば，記号列 t は fa **に受理される** (accept) という.

この型のオートマトンは，**決定性**と呼ばれる. それは，各ステップで，つぎの入力文字が，つぎの状態を一意的に決定するからである.

(3.2.2) 例　図3.2b の状態図に対応する fa は，($\{S, Z, U, V, F\}$, $\{0, 1\}$, M, S, $\{Z\}$) であり，ここで M はつぎのように与えられる：

$$\begin{array}{ll} M(S, 0) = V & M(S, 1) = U \\ M(V, 0) = F & M(V, 1) = Z \\ M(U, 0) = Z & M(U, 1) = F \\ M(Z, 0) = V & M(Z, 1) = U \\ M(F, 0) = F & M(F, 1) = F \end{array}$$ ■

ここでしばらく先に進むのをやめて，これまで学んできたことを復習してみよう. われわれは，規則の右辺が一意的である正規文法から出発した. これから，状態図を組立てることができた. 実際，この図は fa の非公式の表現となっていた. もし文 x が，文法 G に属していれば，これは，G に対応する fa によっても受理されることを調べるのはたやすい. しかし，任意の fa によって受理される記号列が文となるような文法が存在することを示すことは，これ

よりいくらかむずかしい．この結果は，以後使わないので，証明は省略したい．

計算機内部における表現

状態 $S_1, \cdots, S_n$ と入力文字 $T_1, \cdots, T_m$ をもった fa は $n \times m$ 行列 B で表現することができる．配列要素 $B[i, j]$ には，$M(S_i, T_j) = S_k$ となるような状態 S_k の番号 k が入っている．便宜的に S_1 を初期状態とし，最終状態の集合をベクトルで表すことができる．このような行列は状態から状態への推移の方法を示しているので**推移行列** (transition matrix) と呼ばれることがある．

リスト構造で表現する方法もある．k 本の矢印が出ている状態は $2*k+2$ 語を用いて表す．第1語には，状態名が入り，第2語には，k の値を入れる．それに続く語は，2語づつ対になって，1語には入力アルファベットの終端記号が入り，もう1語には，その文字によって移る状態に対応するリスト表現の先頭番地へのポインタが入る．

非決定性 fa

もし G に，つぎのような同じ右辺をもった2個の規則

$$V ::= UT \quad \text{および} \quad W ::= UT$$

が含まれていれば，fa を構成するときに面倒が生ずる．つまりこの場合，状態図の中で U から出る矢印で，T と名札のついたものが2本できて，写像 M が1価ではなくなる．このような図から構成された fa を非決定性と称し，つぎのように定義する．

(3.2.3) **定義**　**非決定性 fa** (nondeterministic fa) あるいは **nfa** とは，5-組(K, V_T, M, S, Z)である．ここでそれぞれの記号はつぎのような意味を持つ：

1. K は状態のアルファベットである．
2. V_T は入力アルファベットである．
3. M は $K \times V_T$ から K の部分集合への写像である．
4. S は K に含まれ，初期状態の集合である．
5. Z は最終状態の集合であり K に含まれる．

ここで fa との重要な違いは，M が，唯一の状態への写像ではなくて，状態の集合（空のこともありうる）への写像であることにある．第二の違いは，複数個の状態が初期状態になってよいことである．写像 M を，前と同様に，$K \times V_T{}^*$ に拡張する．つぎのように定義する．

$M(Q, \varepsilon) = \{Q\}$

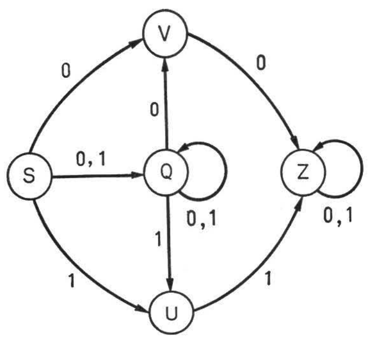
$M(Q, Tt)$ とは $M(Q, T)$ に属する状態 P に対する集合 $M(P, t)$ の合併集合である．ここで T は V_T に属し，t は $V_T{}^+$ に属する．

これをさらに拡張して，集合 $M(P_i, t)$ $(i=1, \cdots, n)$ の合併集合として $M(\{P_1, P_2, \cdots, P_n\}, t)$ を定義する．

記号列 t がオートマトンによって受理されるとは，$M(S, t)$ と Z——最終状態の集合——に属する状態 P が存在することをいう．

例えば，つぎのような正規文法 $G[Z]$ を考えよう：

$$Z ::= U\ 1 \mid V\ 0 \mid Z\ 0 \mid Z\ 1$$
$$U ::= Q\ 1 \mid 1$$
$$V ::= Q\ 0 \mid 0$$
$$Q ::= Q\ 0 \mid Q\ 1 \mid 0 \mid 1$$



(a)

FA= ({S, Q, V, U, Z}, {0, 1}, M, {S}, {Z})

M (S, 0) = {V, Q}
M (S, 1) = {U, Q}

M (V, 0) = {Z}
M (V, 1) = ϕ

M (Q, 0) = {V, Q}
M (Q, 1) = {U, Q}

M (U, 0) = ϕ
M (U, 1) = {Z}

M (Z, 0) = {Z}
M (Z, 1) = {Z}

(b)

図 3.4 状態図と nfa

これをしばらく調べてみれば，この言語 $L(G)$ は，00 または 11 を必ずどこかに含んだ0と1の全ての組合せからなる系列の集合であることがわかる（これを見るには，Z から始めて，文を生成してみよ）．図3.4**a** は，対応する状態図であり，図3.4**b** は，この nfa である．ここでは状態 FAILURE は，記号を全く含まない部分集合 ϕ である．

ここで問題となるのは，各ステップごとに，つぎの入力文字を名札につけた矢印が複数本ありうるので，どの道を選んでよいか迷うことである．各ステップごとに，現在の文形式の把手がどれであるかを知ることはできるが，それを何に還元すべきかがわからない．上の機械で，記号列 01001 を，各ステップでいつも右側の道を行くと決めて，受理する模様を示してみよう．まず，初期状態 S から始める．最初の 0 を読込むと，状態 V また Q のいずれかに移る．ここで Q を採る．つぎの文字は 1 なので，3番目の状態は U か Q である．再び Q を選ぶ．下の図は，解析列を完全に示したものである．

ステップ	現在の状態	入力記号列の残り	可能なつぎの状態	選択された状態
1	S	0 1 0 0 1	V, Q	Q
2	Q	1 0 0 1	U, Q	Q
3	Q	0 0 1	V, Q	V
4	V	0 1	Z	Z
5	Z	1	Z	Z

さて，任意の正規文法から状態図を作り上げそして nfa を作ることができることに注意しよう．この状態図は，複数個の初期状態を持っているかもしれない．G の任意の文がこの nfa によって受理されることは，かなり明らかである．つまり，nfa を動かすということは，上向き構文解析を行っているということのいいかえである．決定性の場合のように，任意の nfa に対してその nfa によって受理される記号列が文となるような正規文法を見出すことができる．これも証明は省略する．

nfa から fa の創成

nfa の難点は，処理の各段で，同じ名札のついた矢印が何本もあるときに，どの矢印を選んでよいかわからないところにある．そこで，nfa が与えられた

らそれから fa を構成する方法を示そう．これは，ある意味で，可能な構文解析を全て並行して行い，不成功に終るものを削除したものといえる．すなわち，この nfa が例えば三つの状態 X, Y, Z のいずれかを選ばねばならないとすると，それに対応する fa は，単一の状態 $[X, Y, Z]$ を持つことになる．これは三つのうちの一つづつを表している．nfa の各ステップで移りうる状態は，この nfa の全ての状態の集合の部分集合であり，相異なる部分集合の数は有限であることに注意しよう．

(3.2.4) **定理** $F=(K, V_T, M, S, Z)$ を記号列の集合を受理する nfa とする．つぎのようにして fa $F'=(K', V_T, M', S', Z')$ を定義しよう：

1. 状態のアルファベットは，K の全ての部分集合からなる．K' の要素を $[S_1, S_2, \cdots, S_i]$ と表わそう．ここで $S_1, \cdots, S_i$ は K の状態である（ただし，状態 $S_1, S_2, \cdots, S_i$ はつねに何らかの意味で正準な順序にならんでおり，例えば，部分集合 $\{S_1, S_2\}=\{S_2, S_1\}$ に対する K' の状態は $[S_1, S_2]$ のみである）．
2. 入力文字の集合 V_T は，F と F' に共通である．
3. 写像 M' はつぎのように定義される：

$$M'([S_1, S_2, \cdots, S_i], T)=[R_1, R_2, \cdots, R_j]$$

ここで

$$M(\{S_1, S_2, \cdots, S_i\}, T)=\{R_1, R_2, \cdots, R_j\}$$

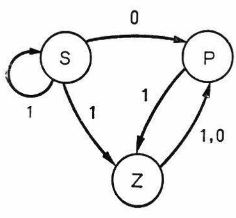
である．

4. $S=\{S_1, \cdots, S_n\}$ とすると，$S'=[S_1, \cdots, S_n]$ である．
5. $Z=\{S_j, S_k, \cdots, S_l\}$ とすると，$Z'=[S_j, S_k, \cdots, S_l]$.

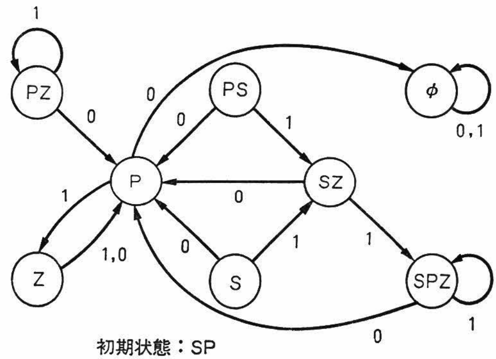
このように定義すると F' によって受理される記号列の集号は，F のそれと同じものになる．証明は読者に任せる．■

(3.2.5) **例** 下図の左側は，ある nfa の状態図である．ここで S と P が初期状態で，最終状態は Z ただ一つである．右側の図は，これに対応する fa である．初期状態は $[S, P]$ であり，最終状態の集合は $\{[Z], [S, Z], [P, Z], [S, P, Z]\}$ である．2個の状態 $[S]$ と $[P, Z]$ は，それぞれに向う矢印がないので，削除してよいことに注意．ここでは，最小の fa――状態数最小の fa――を生成したわけではない．最小の fa を構成するアルゴリズムは

存在するが，これについては，3.6 節にあげる文献を参照されたい.■



初期状態：S，P
最終状態：Z



初期状態：SP
最終状態：Z, ZP, SZ, SPZ

正規表現

この節の以下の部分は 3.5 節 AED RWORD システムを理解するためにのみ必要なものである.

2.9 節で，ある単一の非終端記号に対する（複数個の）規則に対する右辺の系列に現れる記号と超言語記号 |，(,)，{ および } の使用法を説明した. そのような右辺と正規表現には密接な関係がある. $S=\{S_1, \cdots, S_n\}$ とすると，

1. ϕ, ε, $S_1, \cdots, S_n$ は集合 S の上の正規表現である.
2. もし e_1 と e_2 が S の上の正規表現ならば，

$$(\ e_1\ ),\quad \{\ e_1\ \},\quad \{\ e_1\ \}n,\quad e_1e_2,\quad e_1\ |\ e_2$$

も正規表現である. ここで ε は空記号を表し，ϕ は空集合を表す. 正整数 n に対して正規表現 $\{\ e\ \}\ n$ とは

$$\varepsilon\ |\ e\ |\ (e)(e)\ |\ \cdots\ |\ \underbrace{(e)\ \cdots\ (e)}_{n\ \text{個}}$$

と等価な正規表現である. したがって，これについてはここでこれ以上説明する必要はないだろう.

正規表現の値とは，S の上の集合で，$|e|$ と書き，つぎのように定義される：

1. $|\phi| = \phi$, 空集合.
2. $|\varepsilon| = \{\varepsilon\}$（空記号のみからなる集合）
3. $|S_i| = \{S_i\} \quad i = 1, \cdots, n$
4. $|(e)| = |e|$
5. $|e_1e_2| = |e_1||e_2| = \{xy| \quad x \in |e_1| \quad$ かつ $\quad y \in |e_2|\}$
6. $|e_1|e_2| = |e_1| \cup |e_2|$
7. $|\{e\}| = |e|^*$（$|e|$ の閉包）

ϕ 記号を除いては，正規表現の意味は，ある非終端記号に対する規則の右辺の集合の意味と同様である．例えば，$E ::= 0\{+1\}$ と書いて構文要素 E は先頭を0 としその後に $+1$ を任意の回数繰返す記号列を生成することを意味するのに対し，正規表現 $0\{+1\}$ は，同じ記号列の集合を定義している．正規集合 ϕ は，ただ一つの規則 $V ::= V$ だけからなる文法によって記述することができることに注意．何故ならば，これに対応する言語には，一つも記号列が含まれていないからである．

正規表現は語句を記述するのに便利な道具である．(3.1.1) が，立派な例である．さて，正規表現 e に対する集合 $|e|$ を受理するような fa の構成法を示そう．これは，任意の正規文法の言語が，ある一つの正規表現で定義できること，そしてその逆も可能であることを意味している．しかしながら，もっと重要なことは，この fa が，集合 $|e|$ に属する任意の記号列を構文解析する道具であるという事実である．

正規表現に対する fa の構成

正規表現 e に対する fa を以下のような4段のステップで創り出す：

1. e に対する推移システム（以下に定義）を構成する，
2. 推移システムから状態図を構成する，
3. 状態図から nfa を構成する，
4. nfa から fa を構成する．

ステップ 3. と 4. はすでに説明した．そこでステップ 1. と 2. だけを説明すればよい．推移システムを用いることによって，いくつかの正規表現を表している状態図を結び合せる過程が簡単になる．

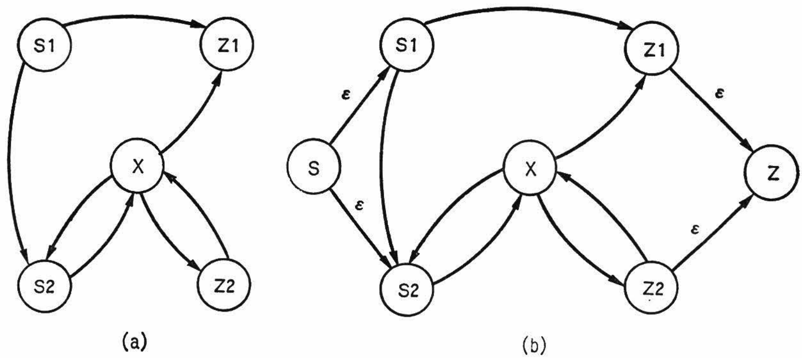


図3.5 状態図とそれに等価な推移システム

(3.2.6) **定義** **推移システム** (transition system) とは，単一の初期状態 S と単一の最終状態 Z をもった状態図である．S に向かう矢印は1本もなく，Z から出る矢印も1本もない．さらに，空記号 ε を名札につけた矢印が存在する可能性がある．■

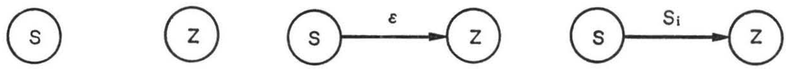
図3.5b は推移システムの例である．これは，**図3.5a** のこれに対応する状態図が受理すると同じ集合を受理する．推移システムを“動かす”大略の考え方は，状態図を動かす考え方と同じようなものである．ここでは，ε と名札をつけた矢印の意味を説明するだけでよい．すなわち，現在の状態が A で，ε と名札の付いた矢印が状態 B に向っているならば，現在の状態は A か B のいずれかであると見なす．つまり，つぎの状態というのは，A または B の後続の状態でありうる．

(3.2.7) **定理** 任意の状態図に対して推移システムが存在する．

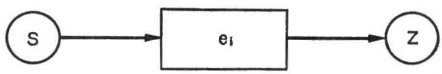
構成法 初期状態が $S_1, S_2, \cdots, S_n$ で最終状態が $Z_1, Z_2, \cdots, Z_m$ である状態図を考えよう．新しくただ一つの初期状態として S を加えて，この状態から $S_1, S_2, \cdots, S_n$ に向かう矢印を出し，それぞれ ε と名札をつける．つぎに最終状態として Z を新しく加えて，$Z_1, Z_2, \cdots, Z_m$ からそこに至る矢印を伸ばして，それぞれに ε と名札をつける．図3.5b は，図3.5a の状態図に対応する推移システムである．■

(3.2.8) **定理** 正規表現 e の各々に対して，$|e|$ を受理する推移システムがそれぞれ存在する．

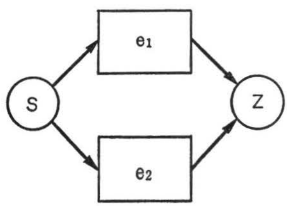
構成法　ϕ, ε, S_i に対する推移システムはそれぞれつぎのようになる.

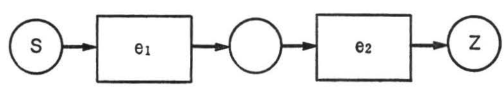


(e) に対する推移システムは, e に対するものと同様である. 今, e_1 と e_2 に対する推移システムがわかっているものと仮定しよう. すなわち, S を初期状態, Z を最終状態として

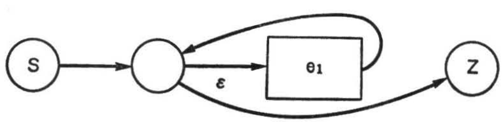


と表すならば, $e_1|e_2$ と $e_1\ e_2$ に対する推移システムはそれぞれ





となる. 上の右の図では, e_1 の最終状態が e_2 の初期状態と一致する. 一方左の図では, e_1, e_2 の初期状態は一致しており, 最終状態も同一のものである. 下の図は, 正規表現 $\{e_1\}$ に対する図である. この推移システムが実際正しいものであるかどうかを確かめることは読者に任せよう. ■



最後は, 推移システムから, 状態図を構成する問題である. ε と名札の付いた矢印を取除かなければならない. つぎに述べるアルゴリズムは, このシステムによって受理される記号列の集合を変えることなしに, 少なくとも1本の矢印を取除くものである (もちろんこれによって, 初期状態や最終状態の数が増えることもあろう). もし, このアルゴリズムを必要な回数繰返せば, ε の付いた枝を全て取除くことができるだろう.

1. 名札 ε の付いた矢印が出ている状態を見付ける．この矢印をたどってつぎの状態に進み，この状態からさらに名札 ε の付いた矢印を探して，この矢印に沿ってつぎの状態に進む．これを，もはや名札 ε の付いた矢印が出ていない状態 B に至るまで続ける：



このような状態が存在しなければ，ステップ3.を実行する．さもなければステップ2.を実行する．

2. もし，初期状態から A(したがってB) に至る名札 ε の付いた矢印の系列が存在すれば，B を初期状態にする．もし B が最終状態であれば，A を最終状態にする．つぎに A から B に至る名札 ε の付いた矢印を取除く．B から C に至る名札 S_i の付いた枝に対しては，A から C に向けて名札 S_i の付いた枝を付加える．これを 図3.6 に示した．

3. ステップ1.に述べた状態が存在しないとすると，$i=1, \cdots, n-1$ に対して状態 A_i から A_{i+1} に至る名札 ε の付いた矢印が存在し，さらに A_n から A_1 に同様な矢印が存在するような，状態の系列，$A_1, A_2, \cdots, A_n, A_1$ が存在しなければならない（証明は読者に任せる)．このときは，これらの状態を全て同一のものと見なす．すなわち全てを単一の状態にする．

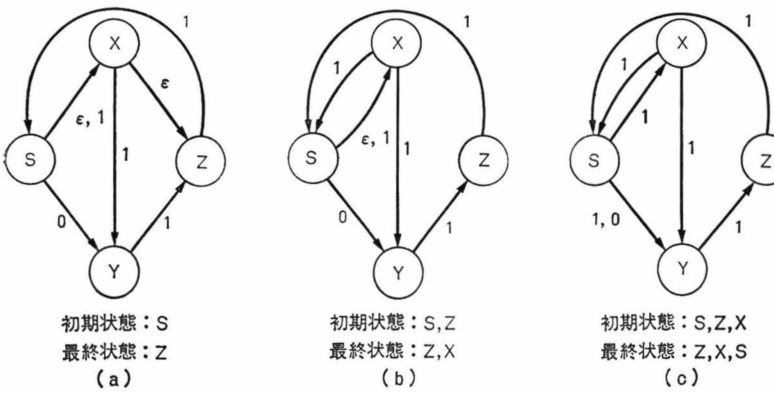


図3.6 ステップ2: 名札 ε の付いた矢印の除去

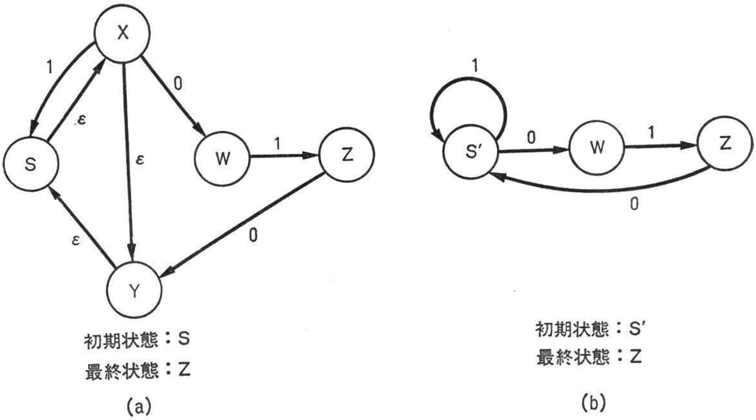


図3.7　ステップ3: 名札 ε の付いた矢印の除去

ここに現れた名札 ε のついた矢印は取除く．もし，これらの状態のうちいずれか1個が初期（最終）状態であれば，新しい状態は，初期（最終）状態とする．これを 図3.7 に示した．

演習問題　3.2

1. 下記の文法 $G[Z]$ に対するオートマトンを構成せよ．これは決定性か．この文法に対する言語を求めよ．

$$Z ::= A\ 0 \qquad A ::= A\ 0\ \mid\ Z\ 1\ \mid\ 0$$

2. 文字1のすぐ右側に必ず文字0があるような0と1からなる記号列を受理する fa を作れ．この言語に対する正規文法を構成せよ．

3. 図3.4 の nfa から fa を作れ．

4. nfa $(\{X, Y, Z\}, \{0, 1\}, M, \{X\}, \{Z\})$ から fa を構成せよ．ただし

$$M(X, 0) = \{Z\} \quad M(Y, 0) = \{X, Y\} \quad M(Z, 0) = \{X, Z\}$$
$$M(X, 1) = \{X\} \quad M(Y, 1) = \phi \qquad M(Z, 1) = \{Y\}.$$

5. 下記の正規表現から nfa を作れ：

a. ((A | B) | { A | C }) | D E M
b. { { 0 | 1 } | (1 1) }

6. 定理 (3.2.4) を証明せよ.

3.3 走査部のプログラム

この節では，小規模で簡単な言語を用いて，走査部をプログラムする方法を示す．われわれが実際に関心があるのは，言語の**記号** (symbol) だけである．走査部が組立てなければならないのが記号だからである．前節までに展開してきた理論を形式的にあてはめるようなことはしない．理論こそは，問題を基礎から理解でき，考察を進める大きな導きになる．

原始言語の記号

ここで取扱う言語の記号は，区切り記号または演算子 /, +, −, *, (,) および // と予約語 BEGIN, ABS, END, それに予約語以外の名前，および整数である．予約語や名前，整数の相互の間は少くとも1個以上の空白で分離する．1個の記号の中に含まれる隣合った文字を空白で分離することは許さない．名前と整数は，つぎの形式をとる：

英字{英字|数字},　数字{数字}.

さらに，走査部は，注釈を取除かなければならない．注釈は，2重区切り記号/* から始まって，初めて2重区切り記号 */ が現れるところで終る．つぎの例は，1個の注釈である．

/* This is * / / a single comment */

走査部の出力

走査部は各記号に対して内部表現を組立てる．多くの場合，これは，固定長の整数である (バイト，半語，1語など)．こうすると以後コンパイラの処理は，可変長の記号をそのまま扱うより実際にはずっと能率よく行うことができる．

この内部表現の中には，“名前”や“整数”を意味する番号も含まれる．すなわち，全ての名前は，同一の内部表現を持つことになる．これは“名前”というものが構文解析部にとっては終端記号であり，どの名前であるかというこ

内部表現	記号	呼び名	内部表現	記号	呼び名
0	未定義	$UND	6	/	$SLASH
1	名 前	$ID	7	+	$PLUS
2	整 数	$INT	8	−	$MINUS
3	BEGIN	$BEGIN	9	*	$STAR
4	ABS	$ABS	10	(	$LPAR
5	END	$END	11	)	$RPAR
			12	//	$SLSL

図3.8 記号の内部表現

とは問題ではないからごく自然な取扱いである．しかし，意味分析部にとっては，名前自体が必要なので，それをどこか適当なところにとっておかなければならない．解決法としては，走査部が2個の値を出力することである．すなわち，第一は，内部表現の番号で，第二が実際の記号そのものか，その記号へのポインタである（もし，走査部が，各種の記号の表を全て管理しており，第二の値として，固定長の整数――表のエントリの番号――を返してくるのならば，処理はますます容易なものになる）．

以後この章では，記号の内部表現を図3.8のように設定しよう．それぞれの記号名に憶え易い呼び名を与えて，これを章を通じて使うことに注意してほしい（ここで走査部を記述するプログラム言語では$記号は文字であって名前の中に使用してよい）．この呼び名は，マクロ定義のようなものを通じて，その内部表現の番号と関連づけられるか，この番号を初期値とする変数を使用することによって，この番号と関連づけることができる．これは，プログラム技法として，非常に貴重なものである．このようにすれば，プログラムがずっと読みやすくなるし，変更も容易になる．

つぎのようなプログラムを考えよう．

```
BEGIN A + / BC // /* COMMENT ++*/ END 11
```

走査部は，これに対してつぎのような出力を呼出し側に返すだろう．

ステップ	出 力	意 味
1	3,'BEGIN'	BEGIN
2	1,'A'	名前 A
3	7,'+'	+
4	6,'/'	/
5	1,'BC'	名前 BC

6	12,'//'	//
7	5,'END'	END
8	2,'11'	整数 11

状　態　図

走査部の作成の手始めとして，記号の構文解析の様子を示す状態図を書いてみる（図3.9）．ここで，いくつか説明を加えておかねばならない．Dという名札は，0, 1, 2, …, 9 という名札のいずれかであることを示す．すなわち，Dは，数字を表している．これは，図を単純にするための処置である．同様にLという名札は，A, B, …, Z を表し，DELIM は，1文字記号 ＋，－，*，（ ，）を表す．/ がここに入っていないことに注意．これは，特別扱いをしなければならない．

矢印のうちには，名札が付いていないものが何本かある．これらは，走査された文字が，矢印の上に陽に現れない場合である．例えば，状態 INT にあって，数字を走査しているかぎりこの状態に留まっている．数字以外の文字を走査すると OUT へ向かう矢印に沿って進む．

OUT と ERROR へ向う矢印はこの状態図では付加的なものである．ここ

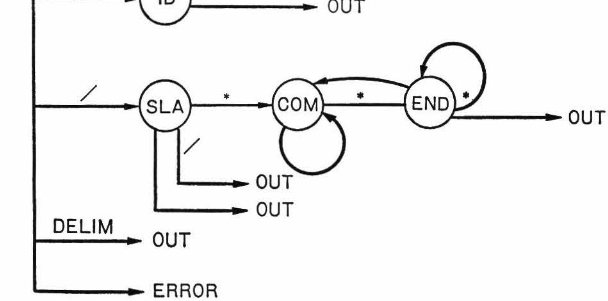


図3.9　記号を走査する状態図

での意味は，走査部によって1個の記号の終りが検出されて，走査部から出てゆくということにすぎない．ここでの問題の一つは，OUT へ出てゆくときに，いま認識した記号のつぎの文字をすでに走査してしまったかまだだったかということである．もし INT または，ID から OUT へ来たのならば，つぎの文字は走査済みであり，区切り記号（DELIM）を認めた後 OUT へ来たのならば，つぎの文字は走査されていない．つぎの記号が要求されたとき，その記号の第1文字がすでに走査済みか否かがわかっていなければならない．これは，何らかのスイッチを設けることによって可能である．ここでは，走査部をでる前に，**必ず**つぎの文字を走査済にしておくというもう一つの方法を採ることにしよう．

さらに，ここでは，状態図として，非決定性状態図よりはむしろ決定性の図を描いた．したがって，/ が $SLASH になるのか，$SLSL になるのか，あるいは，注釈になるのか，これらに共通な状態 SLA を必ず通ることによって，これを決定する問題を配慮した．

この状態図は，形式に従っていないが，記号を構文解析する方法を頭に浮べるのに大変役立った．つぎに，“意味”を付加える方法――構文解析中の記号の取扱い方を知らせる情報を付加する方法――を調べよう．

大域的な変数とルーチン

走査部には，つぎのような変数およびルーチンが必要である：

1. CHARACTER CHAR. CHAR は大域的な変数で，現在走査中の原始プログラムの文字がつねに入っている．
2. INTEGER CLASS. CLASS には，CHAR に入っている文字のクラスを表す整数が入る．ここでは D＝1（数字），L＝2（英字），/ を含む記号のクラス ＝3，DELIM＝4（/ 以外の 1 文字記号）を用いる．
3. STRING A. A は記号を構成している文字列をたくわえている記憶場所を示す．
4. GETCHAR は手続きである．この手続きの目的は，原始プログラムのつぎの文字を走査し，それを CHAR に代入し，そのクラスを CLASS に代入するものである．GETCHAR は，必要に応じて，つぎの入力行を読んで印刷するなどという仕事もしている．上の仕事をするために必要な数命令の写しをそれが必要な全ての場所に置いておくより，このように独

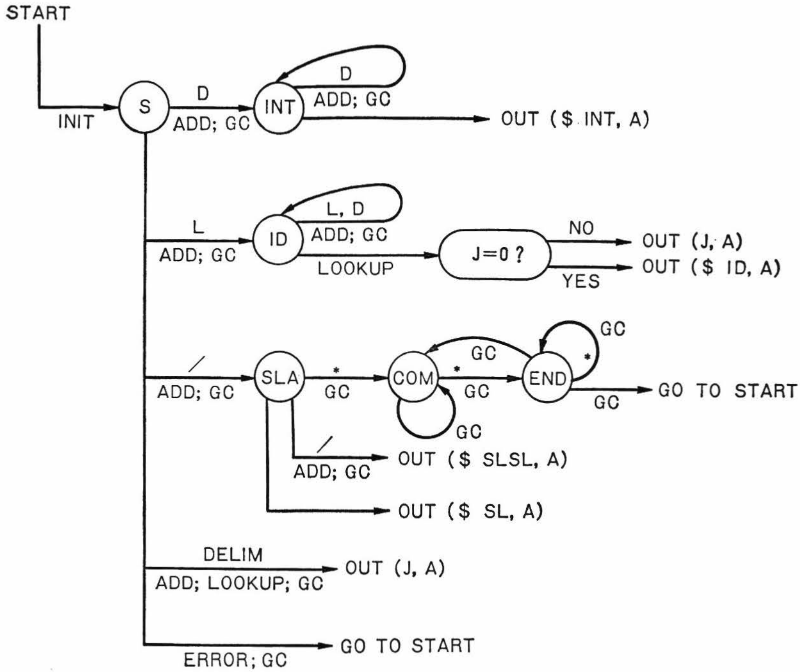


図3.10 意味を追加した状態図

立したルーチンを使うのが良い考えである.

5. GETNONBLANK は, CHAR が空白文字か否かを調べ, もし空白ならば, 空白以外の文字が現れるまで繰返し GETCHAR を呼出すルーチンである.

意味を追加した状態図

記号を構文解析する大様は掴んだので, 記号を組立てるための注意を二, 三追加しよう. これは 図3.10 で行われている. この図は, 本質的には, 図3.9 と変るところはないが, 各枝の下側にそこで実行されるべきコマンドを付加えた. ここには, GETCHAR の略であるコマンド GC も陽に書加えてある. これは, 図3.9 ではかげで行われていたものである. さらに, 名前を認識したあとでそれが予約語か否かを調べる部分を一部付加えなければならなかった. OUT(C, D) という表記は, 走査部を呼出したルーチンに対して, 走査部の出

力として，C，D を返すという意味である．

状態 S に向う最初の矢印の下に INIT コマンドを置いた．これは必要な初期設定を行うものである．われわれの場合は

GETNONBLANK;　　A := ' ';

を行う．これは，CHAR に空白以外の文字が入っていることを確かめ，走査し終った記号を入れるための場所 A を初期設定しておくものである．ADD コマンドは，CHAR にある文字を A := A CAT CHAR によって，記号 A の後ろに連結するものである．LOOKUP コマンドは，A にある記号を，予約語表の中で探索するものである．もし見付かれば，その指標を，大域的な変数 J に代入する．見付からなければ，J には 0 を代入する．

注釈または，誤りが検出されたときは，呼出し側に戻らず，つぎの記号の走査に取掛かることに注意．

記号 // の場合の構文解析のステップを追ってみよう．まず初期設定を行う (INIT) と，状態 S である．CHAR の内容を調べると，それは / であり，SLA に向かう矢印をたどる．矢印をたどりながら，A に / を連結し (ADD コマンド) つぎの文字を走査するために GETCHAR を呼出す．状態 SLA では，CHAR に得られた新しい文字を調べて，名札 / の付いた矢印に進む．矢印に沿って進むうちに，CHAR にある文字を A に連結し，つぎの文字を取出す (GC)． 最後は ($SLSL, A) を返して終る．この記号に続く文字は出口で CHAR の中に入っていることに注意．

読者が自分で，+，235，BEGIM，BEGIN，/* COMMENT */，/ などの記号の構文解析を1ステップづつ行ってみることを奨める．

プログラム

以下の ALGOL 型の手続きは，図3.10 から直接プログラムしたものである．この手続きは，2個の引数 SYN と SEM を持ち，1個は組立てられた記号の内部表現であり，1個は，記号を作っている文字列である．SYN と SEM を大域的な変数にする方法もある．大域的というのは，走査部およびそれを呼出している全ての場所にとってである．こちらの方が一般に効率的で，推奨すべきものである．

この手続きには，CASE 文が含まれている．ここでは，CHAR の中の文字のクラスの番号に基いて，初期状態からどの矢印に沿って進むべきかを決定し

PROCEDURE SCAN (INTEGER SYN ; STRING SEM)

`START: GETNONBLANK; A:=' ';`	この行は初期設定 (INIT).
`CASE CLASS OF`	CASE 文は，状態 S から INT, ID,
`BEGIN`	SLA などへの分岐に用いる.
`BEGIN WHILE CLASS = 1 DO`	CLASS = 1 は数字を意味する. ここは
`BEGIN A:=A CAT CHAR;`	状態 INT である. 数字以外の文字が現
`GETCHAR;`	れるまで文字を走査しそれを A に連結す
`END; SYN:=$INT;`	る.
`END;`	最後に SYN に $INT を代入.
`BEGIN WHILE CLASS ≦ 2 DO`	状態 ID. CHAR が英字なのでここに来
`BEGIN A:=A CAT CHAR;`	た. 英字または数字以外の文字が現れる
`GETCHAR;`	まで走査して，それを A に連結する.
`END; SYN:=$ID;`	
`LOOKUP (A);`	予約語かどうか調べて
`IF J ≠ 0 THEN SYN:=J;`	予約語ならば，SYN に代入.
`END;`	
`BEGIN A:=CHAR; GETCHAR;`	状態 SLA に対応.
`IF CHAR = '*' THEN`	もしつぎの文字が * なら注釈である.
`BEGIN B: GETCHAR;`	
`C: IF CHAR ≠ '*' THEN GOTO B;`	つぎの * が現れるまで走査を繰返す.
`GETCHAR;`	
`IF CHAR ≠ '/' THEN GOTO C;`	* のつぎの文字が／であれば注釈は，こ
`GETCHAR; GOTO START`	こで終り. 新しい記号を読むために，
`END;`	START に戻る.
`IF CHAR = '/' THEN`	
`BEGIN A:=A CAT CHAR;`	ここは SLA から出る名札／の付いた矢
`GETCHAR; SYN:=$SLSL;`	印である.
`END`	
`ELSE SYN:=$SLASH`	これは SLA から出ている名札のない矢
`END;`	印.
`BEGIN A:=CHAR; LOOKUP (A);`	区切り記号の番号を調べて，それを SYN
`SYN:=J; GETCHAR;`	に代入する.
`END;`	
`BEGIN ERROR; GETCHAR;`	許されない文字が現れた. この文字をと
`GO TO START;`	ばして，新しい記号の組立てに戻る.
`END`	
`END; SEM := A;`	CASE 文の終り. 走査部の終り.

ている．

議　　論

このプログラムには議論したい点がいくつかある．第一に，この走査部は，つねに，可能な記号の最長のものを組立てていることに注目されたい．したがって ABC12 は，単一の名前であって，名前のあとに整数が続くものではない．これで，大抵の場合良いのであるが，いつもうまく行くとはかぎらない．有名な例は，FORTRAN の DO 文である．DO10I = 1, 20 で，DO10I は名前ではなくて，記号 DO と文の番号 10 とそれに名前 I である．FORTRAN では，われわれの方法は完全には働かないのは明らかである．そのような場合，普通は，原始プログラムを先まで見て，決定を下すような，小さなプログラムを設計するのが比較的簡単である．

FORTRAN はまた欄が固定されていて，文の番号は，1—5 欄，文は 6—72 欄に書くようになっている．欄の固定された言語に対して走査部を修正するのは簡単であり，ここでは，これ以上述べない．

FORTRAN など 1 枚のカードに 1 文という言語がたくさんある．FORTRAN の走査部は，第 6 欄に空白（または 0）以外の文字がある継続行を見張っていなければならない．これは，GETCHAR ルーチンによれば，非常に容易に解決できる．すなわち，GETCHAR がカードの最後の文字に至ったら，つぎのカードを読んで第 6 欄をまず調べてみる．もし空白（または 0）以外の文字であれば，第 7 欄，第 8 欄，・・・の文字を引続いて返してゆけばよい．もし空白（または 0）であれば，カードの終りを示す特別な文字を走査部に返してやればよい．この文字は，特別な内部記号として走査部から構文解析部に渡される．これらの問題は，このように分離して，最も論理的に容易な場所で解決する方が，プログラム全体で備えるよりは，良い方法である．

プログラム用言語の表現の中には，文字列の中を除いて，空白を全く無視するものがある．これに対しては，GETCHAR ルーチンが，無視する文字を飛ばしてしまうようにすればよい．別法としては，各状態ごとに，そこから発して自分自身に戻る矢印（空白文字を名札につけて）を付加してやればよい．もし同じ GETCHAR をいくつか別々の走査部で使う場合には，こちらの方が良い解決法である．走査部をいくつか使うことはつぎの話題である．

プログラム用言語の中には，それがいくつかの部分に分割できて，それぞれで各記号の構成規則が全く異なっているものがある．これは，例えば，FORT-

RAN の普通の文と FORMAT 文の場合がそうである．ALGOL では，文字列中にある空白は意味があるが，それ以外では完全に無視される．またアセンブリ言語の文をあちこちに埋め込むことを許している言語もある．

このような場合，言語の別々の部分に対してそれぞれ固有の走査部を用意しておき，部分に応じて，あれからこれへの切換え命令（手続き呼出し）を用意しておくのが有利である．これを作るのはいと易いことで，これ以上の議論は必要ない．

GETCHAR ルーチンで文字の CLASS の値を求める最も簡単で能率のよい方法は，ベクトル C を用いるものである．文字の (EBCDIC, ASCII, BCD) 表現を整数として，$C(i)$ に文字 i のクラスの番号を入れておく．これはもちろん，それぞれの文字が，ただ1通りのクラスに分類されると仮定してのことである．IBM 360 の TRT のような命令を使うと，空白でない最初の文字まで走査して，そのクラス番号をレジスタの上に置くことができる．

演習問題 3.3

1. 手続き GETCHAR と GETNONBLANK を，擬ALGOL で書け．
2. 完全な走査部を適当な高水準言語でプログラムしデバッグせよ．

3.4 コンパイラ走査部の自動作成

大部分のプログラム用言語の記号の全般的な構造はかなりよく似ているので，全ての言語に通ずる一般的な走査アルゴリズムを一つ書くことができる．この走査アルゴリズムは，いくつかの表を用いており，それぞれの言語に応じて変っているのはこれらの表だけである．そこで，あるプログラム用言語の記号に関する記述を受理して，その言語用に必要な各種の表を生成する**自動作成機** (constructor) をプログラムすることができる（図3.11）．自動作成機が一たん各種の表を作り上げると，われわれはこれを捨ててしまうことができる．図3.11 では二重線より左側の部分が全て削除できるものである．

先ず，記号の一般的構造を記述することから始めよう (3.4.1 節)．つぎにこの走査アルゴリズムの状態図を描き，このアルゴリズムに必要な表も記述する (3.4.2 節)．最後に自動作成機からみた入力について述べる．

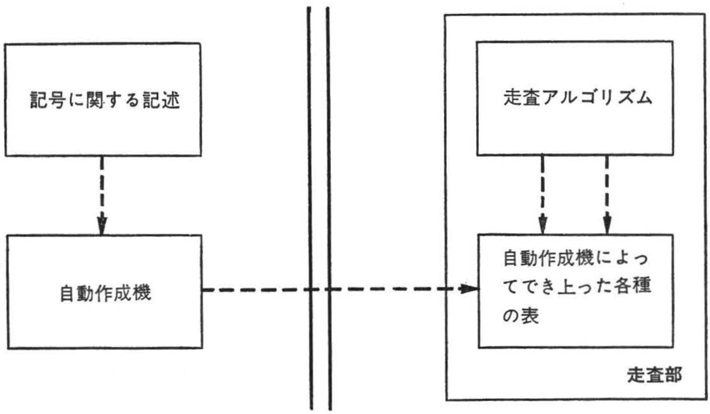


図 3.11　自動作成機と走査部

3.4.1　記号の構造

つぎのような形の記号を構文解析する必要がある.

1. 名前や予約語はつぎの形式である：

   〈先頭文字〉{〈他の文字〉}

   一般に, 先頭文字は, 英字でなければならないが, 残りは英字でも数字でもよい. 名前の文字として $ や _ や − などを認めている言語もある. さらに, 文字数の最大が与えられている場合がある.
2. 整数は〈数字〉{〈数字〉} の形式である. 一般に数字は 0, 1, …, 9 であるが, 8 進数では, 0, 1, …, 7 であり, 16 進数になると 0, 1, …, 9, A, …, F である.
3. /, −, + のような 1 文字記号がある.
4. //, /* のような 2 文字記号がある.
5. ALGOL の 'BEGIN' とか, FORTRAN の .EQ., .AND. などのようなキーワードがある. 場合によっては, これらは, 予約語となって名前に使えないこともある. この場合, 走査と構文解析が大変容易になる.

これ以外にすることは, 空白や同様の文字についてその意味を決めてやることだけである. すなわちこれらの文字は, 完全に無視するのか, 他の記号の区切りに使われるのか, 他の文字と同じ扱いなのか. この走査部の中では, 注釈

の処理は行わない．それは，利用者が走査部から走査部へと行ったり来たりすることができるようにしているからである．したがって，注釈の先頭となる記号を検出すると，利用者は，これを処理するための別の走査部に切換えることができる．

3.4.2 走査アルゴリズム

文字のクラス

構文解析すべき記号の型を調べてみると，つぎのような文字のクラスが必要であることがわかる：

1. DIGIT——“整数”と呼ばれる記号中に現れることができる文字
2. IDBEG——“名前”の先頭文字として現れることができる文字
3. IDCHAR——“名前”の第2文字目以後に現れることができる文字
4. IGNORE——走査部によって，完全に無視されるか，除去される文字
5. INVTERMIN——現在形成中の記号の最後を示す文字，他の場合には，無視されるか，除去される．
6. DELIM——＋，－，：，/，（，）などのようにそれ自体で記号である文字．これらの文字は，2文字区切り記号として使ってもよいし，キーワードを区切るために使ってもよい．

   これらのクラスの大部分は，互に共通部分を持ってはならない——すなわち，これらのクラスの二つの中に同時に現れる文字があってはならない．唯一の例外は，DIGIT と IDCHAR である．われわれのプログラムでは，能率のために DELIM をさらにつぎのような4個のクラスに分割する．
7. 2文字記号やキーワードの先頭に来ない文字．
8. 少くとも1個のキーワードの先頭文字にはなるが2文字記号のそれにはならない文字．
9. 少くとも1個の2文字記号の先頭文字になるがキーワードのそれにはならない文字．
10. キーワードと2文字記号のいずれの先頭文字にもなる文字．

状 態 図

これらのクラスを用いて，図3.12のような状態図を描くことができる．こ

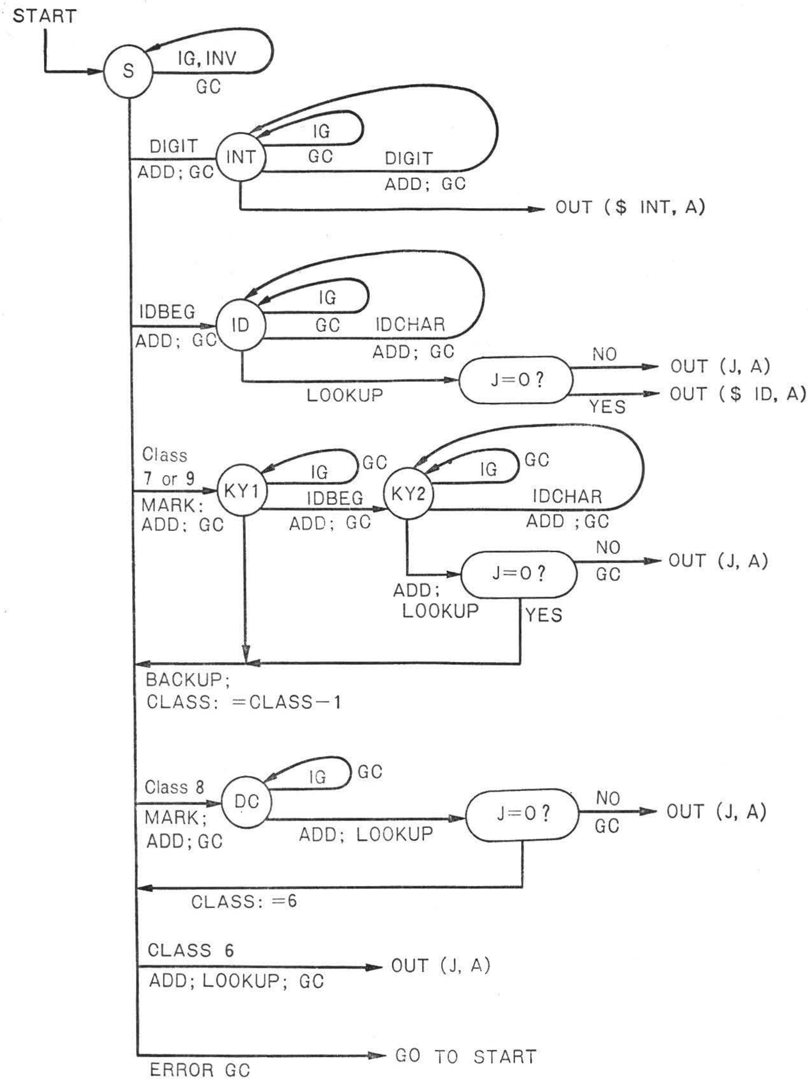


図3.12 走査アルゴリズム

こには，これまでに用いる必要のなかった特徴が一つ加わっている．それは逆戻りである．入力文字列を .ED. としよう．ここで ″.″ は少くとも1個のキ

ーワードの先頭文字である．この文字列は，単一の記号 .ED. なのか，それとも3個の記号 ., ED, . なのか．これは .ED. を記号の一覧表で引いてみるまではわからない．もし .ED. が記号でなければ，GETCHAR ルーチンに，読みすぎた文字を“もとに戻す”ことを依頼する．これでやり直しができる．これには，何らかの方法で，先頭文字の位置にマークを付けておき(MARK)，必要に応じて，GETCHAR ルーチンに最後にマークした位置から再開するために逆戻り (BACKUP) することを頼んでやる．もちろん無視した文字も逆戻りの対象となることに注意．すなわち再び走査するときには必ずしも前の“良い”文字だけを走査するとはかぎらないからである．ある記号を認識することによって，別の走査部に切換わって，同じ文字が先程とは，違った役割をすることもありうるからである．

ここでの逆戻りの性質は限られたものなので，最後にマークした文字だけを憶えておけばよく，そのためにはわずかな時間しか必要でない．記号を一つ出力すると，このマークはもう憶えておく必要はない．

走査アルゴリズム用の表

このアルゴリズムに必要な情報の個々の表現法は，計算機や，アルゴリズムを記述する言語に依存するところが非常に大きいかもしれない．それにも拘らず，ここでいくつかの一般的な基準を示そう．

第一に，言語に現れる全ての記号とそれに対応する内部表現の表が必要となる．この表には，全ての区切り記号，予約語，キーワードも含まれる．LOOKUP コマンドが，ある記号の内部表現を見出すためにこの表を用いる．このコマンドの使用頻度は非常に高いので，この表はハッシュ化されていることが多い（第9章参照）．

必要な表の第二は，状態 S にあるときに，文字のクラス番号を知るために用いるものである．この表の最善の形式は，ベクトルであって，ベクトルの指標としては，文字の EBCDIC 表現などが用いられる：文字 C に対して，V(C) にクラスの番号が入っている．ここで，文字がクラス IDCHAR に属するかどうかは何の役割も演じないことに注意．ここで関心があるのは，IGNORE, INVTERMIN, DIGIT, IDBEG の各クラスだけである．

第三に必要な表は，状態 ID と KY2 で IGNORE, IDCHAR と他のクラスとを区別するために用いられる．

3.4.3 表の自動作成機に対する入力

自動作成機に与えなければならない入力は，文字とその属するクラスおよび，予約語とその内部表現だけである．自動作成機自体は，走査部の定義を読み込み，走査アルゴリズムの中で使われる表を組立てる．走査部を定義する構文はつぎのとおりである：

〈走査部〉 ::= BEGIN 〈名前^1〉 [〈名前^2〉]

　　{〈クラス定義〉} {〈記号定義〉} END

〈クラス定義〉 ::= 〈クラス名〉〈文字の並び〉

〈クラス名〉 ::= DIGIT | IDBEG | IDCHAR | IGNORE

　　| INVTERMIN | DELIM

〈文字の並び〉 ::= [ALLBUT] 〈文字〉 {〈文字〉}

〈記号の定義〉 ::= RES 〈整数〉〈記号〉 {〈記号〉}

〈記号〉 ::= 〈文字〉 {"|" 〈文字〉}

　　| "空白以外の EBCDIC 文字の系列"

〈文字〉 ::= "空白以外の EBCDIC 文字"

　　|〈16 進数〉〈16 進数〉

〈16 進数〉 ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

〈名前^1〉 とは，現在定義中の走査部に与えられる名前で，プログラムがそれを使うときシステムにこの名前を告げる．〈名前^2〉 は，後で説明するサブルーチン名である．

〈文字〉

可能なものならどんな内部表現の文字を使ってもよいので，〈文字〉が示すのは，EBCDIC 表現の文字でも 16 進数を 2 個並べた表現の文字でもどちらでもよいことにしている．空白はつねに，その EBCDIC 表現：X'40' で表されていなければならない．

クラス定義

〈クラス定義〉は，クラス名で与えられたクラスに含まれる文字を書上げる．〈文字〉は全て両側を 1 個以上の空白で区切って互に分離して書かなければならない．ALLBUT を使うとその後に並べた文字と他のクラスに属していない文字以外の全ての文字がそのクラスに属することを示す．ALLBUT は，走査部定義中ではたった 1 回だけ使用することが許される．

3.3 節の言語のクラス定義を例としてみよう：

```
DIGIT         0  1  2  3  4  5  6  7  8  9
IDBEG         A  B  C  D  E  F  G  H  I  J  K  L  M
              N  O  P  Q  R  S  T  U  V  W  X  Y  Z
IDCHAR        A  B  C  D  E  F  G  H  I  J  K  L  M
              N  O  P  Q  R  S  T  U  V  W  X  Y  Z
              0  1  2  3  4  5  6  7  8  9
INVTERMIN     40
DELIM         *  /  +  -  (  )
```

注釈処理のための走査部を定義するとしよう．これは別の走査部が注釈を開始する記号を認めると使用されるものである．注釈の終りの記号が検出されるまで走査を続けるだけでよい．終りの記号を ; としよう．クラス定義は

```
IGNORE ALLBUT ;   DELIM ;
```

となろう．

記号定義

記号の定義はつぎの形式による：

RES 〈整数〉〈記号〉{〈記号〉}

〈整数〉は，上の並びの最初の〈記号〉の内部表現であり，あとに続く〈記号〉には，順にそのつぎに大きい整数が与えられる．記号定義がいくつか現れることもある．RES と〈整数〉，〈整数〉と最初の〈記号〉，そしてそれに続く記号の間は，少くとも1個の空白で分離しなければならない．異なった記号に同じ内部表現を与えることもありうる．

〈記号〉は空白以外の EBCDIC 文字の系列(″RES″, ″|″, ″END″ を除く)であるか，2桁の任意の 16 進数の系列である．これらを記号として用いるためには，″|″ を連接の操作のために使って，2個の〈16 進数〉の系列を内部表現の意味に使う．例えば，記号 ″ RES″ を定義するのに 40 | R | E | S，記号 ″'″ には 7D を用いる．

さらに，それぞれの記号は，つぎの形式のいずれかをとらねばならない：

```
DELIM [DELIM]
```

```
          DIGIT  {DIGIT}
          IDBEG  {IDCHAR}
    DELIM IDBEG  {IDCHAR} DELIM
```

3.3 節の簡単な言語に対する記号定義は以下のようになる：

```
RES   3   BEGIN ABS E | N | D
RES   6   / + - * ( )
RES  12   //
```

サブルーチン呼出し

システム全体をより融通の効くものにするために，原始プログラムの新しい行が読込まれたときはいつでも，使用者に対して，呼出すべきサブルーチン名(〈名前²〉) を明確にしておく．これは，使用者が，その新しい行を見て，必要なら変更が行えるようにするのが目的だからである．これは，欄の固定した書式に適当な内部記号を挿入して自由な書式に変えたり，継続行を調べたり，あるいは注釈行を調べたりするのに使われるだろう．もちろん，このルーチンには，あまり多くの処理をさせるべきではない．多くをさせるとシステム全体の目的——速くかつ自動的な走査部——を損うことになってしまうだろう．

このルーチンは，文字列型の引数を2個持っている．第一は，入力行であり第二は，走査部が実際に処理している行を示す変数の名前である．このルーチンを用いると，走査すべき文字列を，第2引数が設定した変数に移さなければならない．

ここで FORTRAN 処理のルーチンを例にあげよう．読込んだカードが注

PROCEDURE FORT (STRING IN, OUT);	FORTRAN カードの前処理用手続き．
BEGIN	
IF SUBSTR (IN, 0, 1) = 'C' THEN OUT:=X'03'	もし注釈行なら，直前のカードの終りの記号を出力する．
ELSE IF SUBSTR (IN, 5, 1) ≠ ' ' THEN OUT:=SUBSTR (IN, 6, 66)	もし継続行ならば 7–72 欄だけをとり出す．
ELSE BEGIN OUT:=X'03' CAT SUBSTR (IN, 0, 72);	さもなければ直前の文を閉じる記号を出力し，1–72 欄の処理を命ずる．
SUBSTR (OUT, 5, 1):=X'04'; END	第6欄に文の番号の区切り記号を埋込む．
END	

釈行でも継続行でもなければ，まず内部文字 X'03' を出力する．これは，区切り記号として定義され，直前のカードにある文を終了させるのに使われることになる．つぎに第6欄を X'04' に変更する，これは文番号の欄の区切りに使用されることになる．第1欄に C のある注釈行は，すぐ取除いて，それを走査する手間を節約する．継続行に対しては第7欄から72欄までしか使わない．

演習問題 3.4

1. 適当な言語を用いて，汎用の走査アルゴリズムをプログラムしデバッグせよ．

3.5 AED RWORD システム

前節では，制限された4通りの書式

```
        DIGIT  {DIGIT}
        IDBEG  {IDCHAR}
        DELIM  [DELIM]
DELIM   IDBEG  {IDCHAR}  DELIM
```

で書かれた記号を走査するための走査部と自動作成機について述べた．記号の構文が制限されていたので，走査アルゴリズムは，簡単で，能率よく，速いものが作成できた．

AED RWORD (Read a WORD) システム (Johnson ほか (68) 参照) は，コンパイラやインタプリタやオペレーティング・システムなどを生成するための AED—1 システムの重要な部分で，前節とは違った方法をとっている．このシステムは，前節のようにして走査部を構成するが，より一般的なものとなっている．文字のクラスは任意個定義でき，これらのクラスによって，記号に対する任意の構造を，その記号が，適当な fa によって構文解析可能なかぎり，明確にすることができる．さらに，特定の型の記号が組立てられるときにはいつでも，使用者の作ったルーチンを呼出すことができる，ただしこれは，新しいカードを読込んだときだけは呼出さない．このような一般性があるので，コンパイラの走査部だけでなく，ほかにも多くの用途がある．

この一般性のために，これは能率が悪いに違いないと云うようなことはない．Johnson によれば，IBM 7094 に RWORD で作られたある走査部は毎

秒 3000 字を処理したという．一方，慎重に手作りした走査部では，同じ仕事を毎秒 8000 字の割合で処理した．頻発する共通的な記号（名前のような）を扱う新しい技法を追加してこの時間をさらに減らすことを試みている．

RWORD の議論は，自動作成機への入力を調べることから始めよう．記述の統一を図るために，もとの RWORD システムとは，若干表記法に変更があることを断っておこう．

走査部定義の構文

走査部の定義は下記の形式による：

```
BEGIN  {〈文字クラスの記述〉}  END
BEGIN  {〈記号の記述〉}  END
FINI
```

この形式は，前節で述べたシステムのものと非常によく似ている．一般性が入込むのは，文字のクラスと記号の記述の内部である．

文字クラスの記述

クラスの記述には，2 通りの方法がある．第一は，EBCDIC 文字だけをクラスの中に属させるものであり，第二は，文字の実質的な内部表現を与えたいときに用いられるものである．第一の例として

```
LET  =  /ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

と

```
PUNCTUATION  =  A.,; + − = ( ) $ /A
```

がある．最初の例は，クラス LET は，英字 26 文字からなると述べている．後の例は . , ; + − = ( ) $ / の文字をクラス PUNCTUATION に置くことを示している．

二つの例とも，= につづく最初の空白以外の文字が，クラスの文字の区切り記号の役割をしている．この文字が再び現れるまでに，この文字のあとに続く全ての文字（空白を含めて）が このクラスに属する．上例では / と A が区切り記号になっている．もちろん，区切り記号自身はそのクラスに属することはできない．この方法は気が効いている．

クラスを記述する第二の方法の例として

```
XX  =  $21, 22, 25$
```

を挙げる．これは，クラス XX は内部表現が 21, 22, 25 である文字から成ることを示している．区切り記号$を使うと，文字の内部表現が与えられていることを意味する．したがって，この記号を，第一の形式で使ったような区切り記号として用いることはできない．

BEGIN, END, FINI 以外の英字の任意の系列をクラスの名前とすることができる．面白いのは，IGNORE という特別な名前を持った文字のクラスで，これは，走査部が，このクラスの文字が入力中に現れると完全にスキップするか無視するような文字の全体から成っている．これは，前節の IGNORE クラスに対応すのるものである．

記号定義

記号クラスの定義は

〈名前〉(〈整数〉) = 〈正規表現〉 $

という形式をとる．ここで，正規表現は，3.2 節で定義したようなものである．正規表現中の変数は

1. |, (,), ', $ および空白を除く任意の EBCDIC 文字
2. 空記号 ε
3. 文字のクラス名
4. 引用符 ' に続く空白を除いた EBCDIC 文字（これは 1. で使用を認められていない文字を使うための便法である．使いたい文字は引用符の後に続ける）．

二つの例を示そう：

```
PUNCT(5)  =  PUNCTUATION $
BEG(3)    =  " B E G I N " $
ID(1)     =  LET { LET | DIG }5 $
```

第一の例は PUNCT の記号は，全て内部表現が5で クラス PUNCTUATION に属する文字であると述べている．第二の例は，記号 'BEGIN' が内部表現

3であり，第三の例は，最初が英字でその後に英字または数字が 0～5 個続き，内部表現が1であることを示している．特別なクラス IGNORE に属している記号は，決して呼出し側に渡されないで，完全に無視される．このクラスの内部表現は不要なので，定義の形式は

```
IGNORE = 〈正規表現〉 $
```

となる．

文字のクラス IGNORE と記号のクラス IGNORE の違いを注意して見てほしい．つぎのような入力を仮定しよう：

```
BEGIN   IGNORE = / /    LET = /AB/  END
BEGIN   ID(1) = LET { LET } $ END   FINI
```

すると "AB A" という入力列は，空白が無視されるので ABA という1個の名前として構文解析されるだろう．一方，

```
BEGIN   SPACE = / /     LET = /AB/  END
BEGIN   IGNORE = SPACE $
         ID(1) = LET { LET } $ END   FINI
```

を仮定すると，"AB A" は2個の名前 AB，A として構文解析されるだろう．何となれば，空白は実質的に組立てられるが，呼出し側には渡されない記号だからである．

最後に述べる特徴は，走査の際にある記号が組上るとそれに対応するサブルーチンの自動呼出しができることである．つぎの書式で記号のクラスを宣言しておくと

```
〈名前〉(〈整数〉,〈サブルーチン名〉) = 〈正規表現〉 $
```

〈名前〉というクラスに属する記号が組立てられると必ず，〈サブルーチン名〉というサブルーチンが呼出される．このルーチンには，どんな処理をさせてもよい．例えば，名前に結びついたルーチンではそれが予約語かどうかを調べるために表を引くことになろう．この操作は，しかし，前節に述べたシステムのように自動的に行われるわけではない．

多くの場合，入力文字列の構文解析は，それぞれ違ったいくつかの方法が可能である．RWORD では，文字列と一致するもののうちで最長のものを記号としてとることを試みている．従って X123 は1個の名前であって，名前 X

のあとに整数 123 が続くものではない. RWORD でも，あいまいさを解決できない場合がでてくる. このような場合には，エラーメッセージを印刷して処理を中止する.

走査部定義の例

最初の例として，3.3 節の走査部を定義してみよう.

```
BEGIN   SPACE = / /
        LET = /ABCDEFGHIJKLMNOPQRSTUVWXYZ/
        DIG = /0123456789/
END
BEGIN   ID(1,CHECKRES) = LET { LET | DIG } $
        INT(2) = DIG { DIG } $
        SSL(6) = / $  SPLUS(7) = + $  SMINUS(8) = - $
        SSTAR(9) = * $  SLPAR(10) = ( $  SRPAR(11) = ) $
        SSLSL(12, COMMENT) = / *$
        IGNORE = SPACE $
END     FINI
```

サブルーチン CHECKRES は，名前が予約語 ABS, BEGIN, END ではないかを調べて適当な処置をするものとしている. COMMENT は，別の走査手続き（以下に述べる）を持った SCAN を呼出して，注釈を読飛ばして戻ってくる. 別に，文字クラス PUNC = A / + − * ( ) A を定義しておいて，記号のクラス PUNCT(6) = PUNC $ を定義するやり方もあったろう. こうすると，全ての区切り記号が同じ内部表現を持ち，使用者は，それがどの区切り記号かを決めなければならなくなる.

注釈用の走査部はつぎのように定義されよう：

```
BEGIN  IGNCHAR = $00, 01, 02, 03,…, A0, A1,…, FF$ END
BEGIN  IGNORE = IGNCHAR $
       NOEND(1) = *|/ $
       TERMIN(2) = */ $
END    FINI
```

自動作成機

RWORD システムが興味あるのは，3.2 節で展開した形式言語理論を直接適用したところにある. ここでは，この自動作成機の細部まで説明することはしない. ある意味で，この自動作成機は，まさにコンパイラであって，文字ク

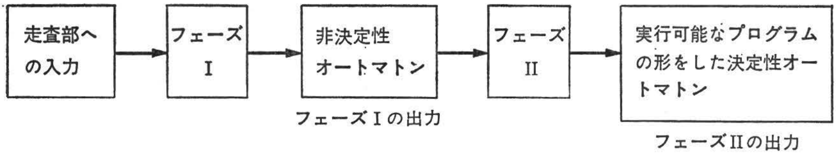


図 3.13 RWORD システムの PART I

ラスと記号クラスを定義する入力情報を読んで記号を認識するプログラムに翻訳するものである.

この自動作成機は, RWORD PART I および RWORD PART II と呼ばれる2段階に別れて機能する. RWORD PART I の大略の流れを図 3.13 に示した. 第1フェーズでは文字クラスを内部的に貯え, 記号のそれぞれのクラスに対する正規表現を構文解析して, それぞれに対して, nfa を生成する. 第2フェーズでは, これらの nfa を全て集めてきて, 1個の決定性 fa を構成する. この決定性 fa は, AED—0 という言語で書かれたプログラムとして出力される. このプログラムは, いくつかの表と, 適当な引数を持った手続き呼出しの系列から成っている. この fa の1個の状態に対して一つの呼出しが対応し, 引数は, 入力文字列に基いて, この状態から他の状態への写像を実質的には記述している.

このプログラムは, コンパイルされ, サブルーチンと結合されて, 実行に入ると, 定義された言語の記号を走査し構文解析を行う. このプログラムは, このままでは各ステップごとにサブルーチン呼出しが多かったり, 表が大きすぎたりして, 能率的ではない. そこで, RWORD PART II でさらに処理が行われる.

手短かにいえば, RWORD PART II は, 時間と空間に関して, 能率のよい走査部を作り出すために必要とされるもので, ある種の最適化が施される. これについては, ここではこれ以上述べない. このほか, いくつかの走査部が, 一しょにまとめられて (これらが, 同じ仕事で用いられるので) 一つの大きな総合的な走査部になることもある. RWORD PART II の出力はアセンブリ言語のマクロ呼出しの形式による表であって, すでに定義してあるマクロのライブラリを用いてアセンブルしてやらなければならない. これによって, ある程度まで, 機械に依存しないものにすることができる. 何となればマクロ定義は, 別の機械のための表を生成するようにいつでも書換えることができるからである.

3.6 参考文献概説

有限状態機械の概念は，McCulloch と Pitts (43) に由来する．本章の形式化は Moore (56) が示したものと同様なものである． 正規表現とオートマトンとの関係は Kleene (56) に述べられている．もちろん，本章では，オートマトン理論のうわべをかすめたにすぎないので，読者は，Gill (62)，Ginsburg (62)，Hopcroft と Ullmann (69) らの教科書を参照されたい．

3.4 節に述べた自動作成機は，Feldman (66) にある FSL（コンパイラ作成システム）で用いられているものを発展させたものである．AED RWORD システムは，1966 年以来動いており（Johnson ほか (68) 参照），有限オートマトンの理論を字句解析に応用した最初のものである．

第 4 章

下向き構文解析法

本章では，文脈自由文法に対する構文解析法の手始めとして，**下向き** (top-down)，**予測的** (predictive) あるいは**ゴール向き** (goal oriented) などと呼ばれる認識法を導入する．このような呼び方は，これらの認識法が働いて構文の木を組立てる方法によるものである．4.1 節では，この方法を全般的に述べ，この方法の処理時間が本質的に長くなることを述べる．その理由は，逆戻りの回数を多く必要とするためである．そこで，ある種の文法に対して逆戻りの回数を減らすようなアルゴリズムをついで与える．4.2 節では，下向き解析法に伴う諸問題を論じ，規則の並べ方を変更する方法を概観し，種々の難点を克服するために文法の表記法を変えてみる．計算機内での文法の表し方も議論する．最後に 4.3 節では，再帰的手続きを用いて下向き解析法をプログラムすることに言及する．

4.1 逆戻りのある下向き解析

下向き構文解析アルゴリズムによる構文の木の組上げは，2.5 節図 2.7 に示したように木の根から始めて，文に向けて下向きに行う．この例で示したように，基本的な考え方は単純かつ直截的である．複雑になる主たる原因は，逆戻りして可能な木を確かに全て試すことができるために情報を管理しなければならないことにある．このような面倒な事態を最小限に留めて，このアルゴリズムを説明するのに，構文の木を組立てる仕事をする人になぞらえることにしよう．構文解析の任意の時点で，それまでにでき上った部分木の各点に 1 人ずつ人を立たせておく．端点にいる人は，解析される文の記号の上に立っていることになる．

ある人が，文 x の構文解析を命ぜられる．彼は，Z を出発記号として，

$Z \overset{+}{\Rightarrow} x$ という導出を見付けにゆく仕事を始める．まず，$Z ::= y$ という規則があれば，直接導出は $Z \Rightarrow y$ でなければならない．Z に対する規則が

$$Z ::= X_1X_2\cdots X_n \quad | \quad Y_1Y_2\cdots Y_m \quad | \quad \cdots \quad | \quad Z_1Z_2\cdots Z_l$$

であるとしよう．彼は，まずこれらの規則のうちから $Z ::= X_1X_2\cdots X_n$ を選ぶ．この規則を使って，構文の木を組立てることができなければ，彼は，つぎの規則 $Z ::= Y_1Y_2\cdots Y_m$ を選んでくる．これでもうまくゆかなければ，またつぎの規則を試してみるというように繰返す．

では，彼は直接導出 $Z \Rightarrow X_1X_2\cdots X_n$ が正しいものかどうかどのようにして調べたのだろうか．もし正しいとすれば，$X_i \overset{*}{\Rightarrow} x_i$ $(i=1, \cdots, n)$ となるような記号列 x_i が存在して，文 $x = x_1x_2\cdots x_n$ となるだろう．彼は最初に，息子 M_1 をつれてきて，$x = x_1\cdots$ となるような任意の x_1 に対して，$X_1 \overset{*}{\Rightarrow} x_1$ となる導出を見付けることを命ずる．もし M_1 がこの導出を見出すことができたら，彼（と彼の息子たち，孫たち…のだれも）が文 x に含まれる記号列 x_1 をカバーし尽したので，M_1 は父にうまくいったことを報告する．父はつぎに，2番目の息子 M_2 を呼んで $x = x_1x_2\cdots$ となるような x_2 に対して $X_2 \overset{*}{\Rightarrow} x_2$ となる導出を見つけることを命じて，息子の報告を待つ．このようにして，息子 M_{i-1} が，成功を報告してくるかぎり，つぎに息子 M_i に $X_i \overset{*}{\Rightarrow} x_i$ という導出を見出すことを命ずることをつづける．もし息子 M_n が成功を伝えてくれば，構文解析列が見出されたことになる．

しかし，もし息子 M_i が $X_i \overset{*}{\Rightarrow} x_i$ を見付けることができないならばどうなるだろう．M_i は，うまくいかなかったことを父に伝えると，父は M_i を追い出して（勘当して），M_i のすぐ上の兄 M_{i-1} につぎのように命ずる「おまえは，すでに導出が見つかったといっていたが，あれは間違っていた．別の導出を見つけて来ておくれ」．もし M_{i-1} が別の導出を見出すことができれば，彼は再び成功を報告し，先程と同じように先へ進むことになる．しかし M_{i-1} が失敗の報告をすると今度は M_{i-1} が勘当されて，彼のすぐ上の兄 M_{i-2} が呼びだされて，もう一度やり直しを命じられる．このように逆戻りして，もし M_1 も勘当されることになると，実は，直接導出 $Z \Rightarrow X_1X_2\cdots X_n$ が間違っていたので，最初の男は，つぎの導出 $Z \Rightarrow Y_1Y_2\cdots Y_m$ を試してみる．

息子達 $M_1, \cdots$ はどんな働きをしているのだろう．息子 M_i の目標 X_i が終端記号であるとしよう．入力系列は $x = x_1x_2\cdots x_{i-1}T\cdots$ という形式であり，$x_1, x_2, \cdots, x_{i-1}$ 中の記号はすでに，別の人々によって調べがついてい

る．M_i は，つぎの未調査の記号 T が自分の目標 X_i と同じかどうかを調べる．もし同じなら，彼はこの記号をカバーしたので成功を報告する．同じでなければ失敗を告げる．

もし M_i の目標 X_i が非終端記号であれば，彼は自分の父がやったと同じように息子達に命じることになる．すなわちこの非終端記号を左辺とする規則の右辺を調べることから始めて必要に応じて彼の息子を呼んで命じたり，勘当したりする．もし彼の息子全員が成功を報告してくれば，M_i は，自分の父に成功を告げることができる．もし彼の父が，別の導出を見つけることを命じて，その目標が終端記号であれば，M_i は不成功を報告する．何となれば，そのような導出は存在しないからである．非終端記号であれば，M_i は彼の息子の一人に別の導出を見出すことを命じて以前と同様に息子からの報告を待って行動する．もし彼の息子達の答が全て失敗であったら，M_i は父に失敗を報告する．

読者は，この方法が何故，予測的とかゴール向きとか呼ばれているかわかったであろう．この方法は，また，下向きとも呼ばれている．それは，構文の木が組立てられる方向を示しているからである．出発記号から始めて下向きに文へと至るのである（図4.1）．

この方法（あるいは説明法）の利点は，節点上の各人が憶えておかなければならない情報が，自分自身の目標と，父親と，息子達と，文法上で自分の置かれている位置と，それに入力記号列とだけである，という点である．だれもよそで何が行われていようと全く知る必要がない．これは，一般に，われわれがプログラムをする際につねに努めてきたやり方である．各プログラム片やサブルーチンは，自分自身に与えられた入力情報と出力情報だけに関わっていればよく，ほかには関与しない．プログラムの中で息子を呼んだり，勘当したりす

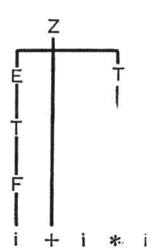
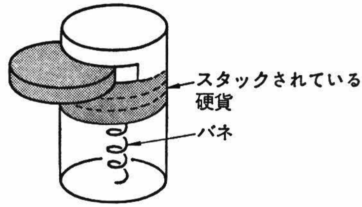


図4.1 i＋i∗i の下向き構文解析の途中



ることを真似るために，**LIFO スタック**(last-in fist-out, **後入れ－先出し**) あるいは**プッシュダウン記憶** (pushdown store) と呼ばれているものを使う．スタックは，大部分の構文解析法で用いられている基本的な機構である．実際，再帰的呼出しを語る時，つねにスタックが引き合いに出される．スタックというのはデータをしまう記憶装置である．ところが，データは最上段の入口からしか入れることができず，すでにそこにあったデータは“押し下げ”られる．したがって，参照したり変更したりできるのは，最上段（または上から数段目まで）の要素にすぎない．もはやしまっておく必要がなくなれば，最上段の要素が取除かれて，その下につづく要素が，押し上げられてくる．日常生活でのスタックの例に，硬貨入れ（上図）がある．硬貨を1枚押し込むと，すでに入っていた硬貨が押し下げられる．そして，一番上にある硬貨だけが取出せるようになっている．

スタックを計算機で実現する普通の方法は，配列 S とカウンタを利用するものである．もし $v=0$ であれば，スタックは空である．もし $v=n$ ($n>0$) であれば，スタックには，$S(1)$, $S(2)$, $\cdots$, $S(n)$ が入っており，$S(n)$ が最上段の要素である．

われわれのアルゴリズムをもっとはっきり示すために，これを擬 ALGOL で書いてみよう．まず第一に，文法は，1次元配列 GRAMMAR の中に並べられ，規則の組 $U ::= x|y|\cdots|z$ を $Ux|y|\cdots|z|\$$ のように記憶しているとしよう．すなわち，各記号はそれぞれ 1 記憶場所に入り，U の右辺はそれぞれ | を後につけ，最後の右辺の後には |$ を置く．したがって文法

(4.1.1)	Z	::=	E #
	E	::=	T + E | T
	T	::=	F * T | F
	F	::=	(E) | i

は，つぎのようになろう．

(4.1.2)　ZE #|$ ET + E|T|$ TF * T|F|$ F(E)|i|$

スタックの各要素は一人の男を表し，５個の成分

(GOAL, i, FAT, SON, BRO)

からなっている．これらはつぎのような意味を持つ：

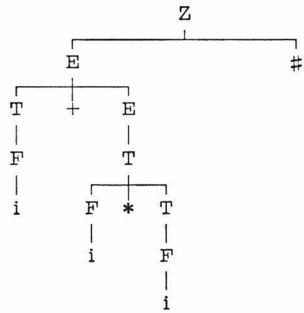
1. GOAL は，この男が試みなければならない目標の記号である．すなわち，彼は GOAL に還元されるべき文の，現にカバーされていない部分の頭部を見出して，これをカバーしなければならない．この GOAL は，彼の父親が彼に与えたものである．
2. i は配列 GRAMMAR の指標であり，現在，この男が働いている目標 GOAL の規則の右辺にある記号を指している．
3. FAT は，この男の父親の名前である(彼の父のスタックの要素の番号)．
4. SON は，彼が最も新しく呼出した息子の名前である．
5. BRO は，彼の直ぐ上の兄の名前である．

これらの値が0であるとそれがないことを示す．プログラム中で v の値は，仕事をしている人の数（スタック要素の現在数）を示し，c は現在働いている人の名前（スタック要素の番号）を示す．彼以外の人はみな彼の仕事が終るのを待っている．指標 j は入力記号の系列 INPUT(1), ···, INPUT(n) のうち構文の木にカバーされていない最も左側の記号を指している．

このプログラムで，GOAL, i, FAT, SON, BRO は，もしよそで値がきめられなければ，現在仕事をしている男（名前は c）の各成分を参照している．これは，プログラムを読むときに忘れてはならない重要な点である．例えば，S(c) の GOAL 成分を参照するのに，S(c).GOAL と書くべきで GOAL ではない．この省略形を用いて来たのはプログラムがごたごたするのを避けるためである．

働いている各人が，自分の息子達を憶えておかなければならないことを先に述べた．一つの方法は，自分の息子達全員の名前を彼のスタック要素に留めておくことであるが，これには，SON に対して可変個の欄を要することになる．そこで，ここでは，SON の欄に，最も年少の息子へのポインタを置くことにする．この息子の BRO 欄をみると，彼の直ぐ上の兄へのポインタがあり，この兄の BRO 欄も同様になっている．

これを説明するために，文法 (4.1.1) の文 i+i*i に対する 図4.2**a** の



(a) **構文の木**

STACK	GOAL	i	FAT	SON	BRO
1	Z	4	0	15	0
2	E	10	1	7	0
3	T	20	2	4	0
4	F	28	3	5	0
5	i	0	4	0	0
6	+	0	2	0	3
7	E	12	2	8	6
8	T	18	7	12	0
9	F	28	8	10	0
10	i	0	9	0	0
11	*	0	8	0	9
12	T	20	8	13	11
13	F	28	12	14	0
14	i	0	13	0	0
15	#	0	1	0	2

(b) **構文解析後のスタック**

GRAMMAR = ZE#|$ET+E|T|$TF*T|F|$F(E)|i|$

図4.2　i+i*i の下向き構文解析終了時のスタック

構文の木を考えよう．構文解析アルゴリズムの終了時点のスタックの様子を図4.2**b** に示した．ここで人2（S(2)）は目標 E を持っている。彼は，構文の木によれば，E ::= T + E という規則を使用するとされている．したがって彼は，T，+，E を見出すために 3 人の息子を従えていなければならない．S(2).SON = 7 であるから，彼の末の息子は人7であり，人7の目標はEである．下から2番目の息子は S(7).BRO であり，これは 6 である．人6の目標は + である．長男の名前は S(6).BRO にあり，これは 3 である．

このように各人の息子の 1 次元リストを示すことができるが，このリストは，スタックの中でいわゆる“珠数つなぎ”になっている．このような珠数つなぎにしたリストを使うのは，プログラミングの常道である．完全な理解ができない読者はこの例をよく勉強して，このようなリストに慣れてほしい．このスタックが，与えられた文に対する構文の木の内部表現になっていることは明らかであろう．

文法 (4.1.1) によれば，各文が区切り記号 # で終っていることに注意しよう．この記号を見て，文全体が構文解析されたのであり，まだ，文の頭部が終ったにすぎないのではないことが確かめられる．

下向き構文解析アルゴリズムを以下に示す．これは，機能の異なるものを分離すると6部分に分割された．余り重要でない部分を細部まで記述してはかえって明解さを欠くことになるので，そこは普通の言葉で記述した．

INITIALIZE

S(1):=(Z, 0, 0, 0, 0); c:=1; v:=1; j:=1; GO TO NEW MAN	最初の男を呼出す．彼の目標はZすなわち出発記号である．

NEW MAN

IF GOAL は終端記号である THEN IF INPUT(j) = GOAL THEN BEGIN j:=j+1; GO TO SUCCESS END	新しい男に目標を与える．それが終端記号のとき，GOAL が文の記号と一致するならば，成功を報告する．
ELSE GO TO FAILURE;	一致しなければ失敗を報告する．
i:=GOAL の最初の右辺の GRAMMAR 中 の指標; GO TO LOOP	新しい男の目標が非終端記号の場合，GOAL の規則の右辺への指標を設定．

LOOP

IF GRAMMAR(i) = "|"	右辺を見る．
THEN IF FAT ≠ 0	右辺の区切りであれば，
THEN GO TO SUCCESS	成功を報告．
ELSE STOP 一文が得られた;	父がなければ，文が得られているので終り．
IF GRAMMAR(i) = "$" THEN IF FAT ≠ 0	試みるべき右辺が尽きたので，
THEN GO TO FAILURE ELSE STOP 一文は得られず;	失敗を報告．父がいなければ文が得られないまま終了．
v:=v+1; S(v):=(GRAMMAR(i), 0, c, 0, SON);	GRAMMAR(i) が試みるべきもう一つの目標である．新しい息子を呼ぶ．c が彼の父でありSON が彼のすぐ上の兄である．
SON:=v; c:=v; GO TO NEW MAN;	SON と c を更新して新しい息子に着目し彼の報告を待つ．

SUSCCESS

c:=FAT; i:=i+1; GO TO LOOP;	父親に成功を報告し父はつぎのステップへ進む．

FAILURE

c:=FAT; v:=v−1; SON := S(SON). BRO; GO TO TRY AGAIN;	父親に失敗を報告する．父はこの息子を勘当し，この息子のすぐ上の兄にもう一度試行を命ずる．

TRY AGAIN

IF SON = 0 THEN BEGIN WHILE GRAMMAR(i) ≠ "|" DO i:=i+1; i:=i+1; GO TO LOOP END;	再度試行をさせるべきすぐ上の息子はいるか．いなければ，この右辺はスキップして——これは望むものではない——新しい右辺の試行に進む．
i:=i−1; c:=SON; IF GOAL は非終端記号である THEN GO TO TRY AGAIN; j:=j−1; GO TO FAILURE	すぐ上の息子がいる場合．彼に再試行を命ずる．彼の GOAL が非終端記号であれば，成功を求めて再試行に進む． 彼の GOAL は終端記号である．彼は再試行できない．彼は与えられた記号をカバーできず失敗を報告する．

このアルゴリズムは一見良さそうにみえるが，重大な欠陥を持っている．これは次節で説明しよう．まず，このアルゴリズムを修正して，逆戻りの回数を減らすようにしよう．だれかが失敗を報告してくると，彼をすぐ勘当して，彼の直ぐ上の兄に“再試行”——彼の目標に還元される別の部分記号列を見出すように——を命じていた．このようにするのは，息子をほとんど信用していないことを示している．もし息子を呼出す前に慎重に検査を行って，息子を正しい方向に向わせれば，もう少し，彼に信頼がおけるようになるだろう．われわれは，息子は1回目は，間違いなく仕事を遂行するものと仮定するのである．したがって，1回目は，再試行を命ずる理由はない．だれかが失敗したことがわかったときに間違った道をたどってきてしまったと考えてここで全く別の規則（規則の右辺）を試みるようにする．

これを実現するために，1回目には正しい規則が選ばれるように各非終端記号に対する規則を並べ直してやらなければならない．大抵のプログラム用言語では，これは自然に行えるが，自動的にこれを行う方法はない．ここでは，最も長い選択枝を最初にもってくるようにする．文法 (4.1.1) は，すでにこのように並べてある．したがってもし，人が目標 E を持っていれば，彼はまず

T＋E を試みる．もし T は成功するが ＋ がだめな場合，彼は彼の長男に別の T を見つけてくることを再び命じることをしないで，彼の息子全員を勘

INITIALIZE

S(1):=(Z, 0, 0, 0, 1); c:=1; v:=1; GO TO NEW MAN	最初の男を呼んで目標 Z を与え，最初の記号から始める。

NEW MAN

i:=GOAL の最初の右辺の GRAMMAR への指標； GO TO LOOP	非終端記号のときだけ息子を呼ぶ．息子は，自分の規則を探して，働きに出る．

LOOP

IF GRAMMAR(i) = "|" THEN IF FAT ≠ 0 THEN GO TO SUCCESS ELSE STOP－文が得られている；	右辺を見る． 右辺を調べ終っているので成功を報告する．もし父親がいなければ文が得られて終了．
IF GRAMMAR(i) = "$" THEN IF FAT ≠ 0 THEN GO TO FAILURE ELSE STOP－文ではない	試みるべき右辺が尽きた． 父親に失敗を報告． 父がいなければ，文が得られずに終了．
IF GRAMMAR(i) は終端記号である THEN IF INPUT(j) = GRAMMAR(i) THEN BEGIN j:=j+1; i:=i+1; GO TO LOOP END ELSE GO TO TRY ALTERNATE;	終端記号が一致していれば，この記号はカバーされたのでつぎの記号に進む． 終端記号が一致していなければ別の右辺を選んで試みなければならない．
v:=v+1; S(v):=(GRAMMAR(i), 0, c, 0, j); SON:=v; c:=v; GO TO NEW MAN;	つぎの息子を呼んで非終端記号を GOAL とする．この息子の父は c で示す． SON と c を更新して，注意を息子に向け息子の報告を待つ．

SUCCESS

c:=FAT; j:=S(SON). j; i:=i+1; j:=j+1; GO TO LOOP	父親に成功と，どこまでカバーしたかを告げる．そして，つぎの規則の記号とカバーされていない最初の記号に進む．

FAILURE

c:=FAT; v:=c; SON:=0; GO TO TRY ALTERNATE	父親に失敗を報告する．父親はこの息子とその兄弟全員を勘当し別の選択枝を試みる．

TRY ALTERNATE

IF FAT ≠ 0 THEN j:=S(FAT).j ELSE j:=1; WHILE GRAMMAR(i) ≠ "|" DO i:=i+1; i:=i+1; GO TO LOOP	父親から，どこからカバーをはじめればよいかを知らされて，つぎの選択枝があれば，その先頭を見つけ試行を再開する．

当して，つぎの規則　E ::= T　に進む．

このように変更すると，スタックから BRO 成分が要らなくなるが，一方，記号列のどこまでカバーされどこからカバーされていないか目印をつける必要がでてくるだろう．そこで，新しいスタックの要素は

(GOAL, i, FAT, SON, j)

のようになる．ここで GOAL，FAT，SON は前と同じであるが，j は，INPUT 中への指標で，この場所を，我々自身が現に見ているか，この場所で，報告を待っている．このようにすると，問題全体が単純化されて，右辺に終端記号が含まれているかどうかは，自分で検査できるようになるので，この仕事をやらせるために息子を呼び出す費用を節約することができる．

この新しいアルゴリズムでも，正しい規則が最初に試みられるように規則が並べてあるときにのみ（逆戻りしないで）正しく働くことに注意されたい．

4.2　下向き解析法の問題点とその解決

直接左再帰性

前節のアルゴリズムには，根本的な欠陥がある．それは，左再帰的な GOAL を使用している点である．もし GOAL が X で，X の最初の規則が X ::= $X\cdots$ とすると，直ぐ息子を呼んでつぎの X を試みなければならない．息子も直ちに彼の息子をつぎの X を試みさせるために呼ぶ．このようにして，

だれもが自分の息子に仕事を押しつけて行くので，中国大陸の全人口がかかってもこの問題を解くには不十分である．

文法 (4.1.1) の規則が左再帰的でなくて，右再帰的に書いてあるのは，この理由である．このような直接的な左再帰性を取除くもっと良い方法は，2.9 節に述べた繰返しと選択の記法を用いて規則を記述することである．規則を

(4.2.1) $E ::= E + T \mid T$ に対して $E ::= T\{+T\}$
$T ::= T * F \mid T/F \mid F$ に対して $T ::= F\{*F \mid /F\}$

と書く．ここで直接左再帰的な部分を含んだ文法規則を，繰返しを用いたこれと等価な文法規則に変換する二つの規則を与えよう．

(4.2.2) 括り出し．$U ::= xy|xw|\cdots|xz$ という規則が存在するときはいつでも，これを $U ::= x(y|w|\cdots|z)$ に置換える．ここで丸かっこは超言語記号である．

この括り出しは，算術式で行うと同じようにもっと一般的に適用できる．例えば，(4.2.2) で $y = y_1y_2$，$w = y_1w_2$ であれば，$U ::= x(y|w|\cdots|z)$ は

$$U ::= x(y_1(y_2|w_2)|\cdots|z)$$

と書換えることができる．もし，もとの規則が $U ::= x|xy$ であれば，これを $U ::= x(y|\varepsilon)$ と変換できる．ここで ε は空記号である．このときは，ε はつねに最後の選択枝にする，何故ならば，GOAL が ε であれば，これはつねに一致するという約束を採用するからである．

括り出しを用いると，直接的な再帰性を除去できるばかりでなく，文法の大きさを減じて，より能率的な構文解析ができるようになる．これは後で触れよう．

一たん (4.2.2) の規則が適用されると，ある非終端記号に対して，直接左再帰的な右辺が高々1個存在する可能性がある．それが1個であれば，つぎのような処置を採る．

(4.2.3) $U ::= x|y|\cdots|z|Uv$ を左再帰的な右辺が最後にある規則としよう．上の規則は，x, y, $\cdots$ または z の後に0または1個以上の v が続くものがこの構文のクラスの要素であることを示している．このような規則は
$U ::= (x|y|\cdots|z)\{v\}$ に変換する．

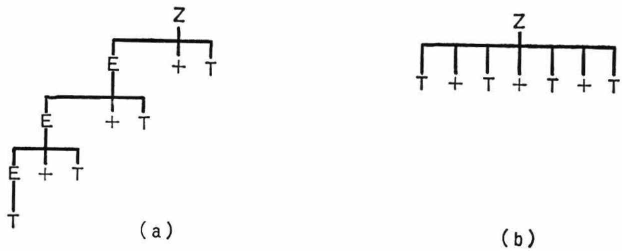


図 4.3　再帰性を用いた木と繰返しを用いた木

われわれはすでに (4.2.1) の変換で (4.2.3) を使っていた．ここで T の前後にある不要なかっこは取除いてあった．別の例として $A ::= BC|BCD|Axz|Axy$ を変換してみよう．

規則 (4.2.2) により $A ::= BC(D|\varepsilon)|Ax(z|y)$，規則 (4.2.3) により $A ::= (BC(D|\varepsilon))\{x(z|y)\}$ となる．不要なかっこは取除くことができて，$A ::= BC(D|\varepsilon)\{x(z|y)\}$ となる．

このように変更すると，下向き構文解析アルゴリズムも変更しなければならないのはもちろんである．このアルゴリズムは，右辺ばかりでなく，右辺の部分記号列に対する選択枝の取扱いも可能でなければならないし，空記号列 ε の取扱いもできなければならない．さらに繰返しの取扱いもできなければならない．これらの変更は読者に任せよう．

再帰性の代りに繰返しを用いると，これまでの木構造は幾分変えなければならない．したがって 図 4.3 **a** は 図 4.3 **b** のようになる．これらの二つの木は同等なものとして取扱われるべきもので，加算演算子は左から右に評価されるものとする．

一般的な左再帰性

まだ左再帰性に関する全ての問題を解決したわけではない．直接的な左再帰性は取除いたが，一般的な左再帰性は残っている．すなわち

$$U ::= Vx \quad で \quad V ::= Uy|v$$

のような規則から $U \stackrel{+}{\Rightarrow} Uyx$ が生成される．これを取除く簡単な方法はないが，その有無を検査することはできる．もとの文法をもってきて，直接的な左

再帰性は全て除去する．するとそこで得られた文法の記号 U が左再帰的であるのは，U FIRST$^+$ U が成立ち，そのときに限られる．これがわれわれの知っている検査法である．

計算機内での文法の表し方

下向き構文解析法の問題点の一つに，計算機内での文法の表現法がある．4.1 節で使われたような方法ももちろんその一つである．非終端記号に対する規則を見出すのに要する手間を考えるとこの表現法が良くないのは明らかである．別の表現法について考えよう．その前に，ある文法を受理して今述べたばかりの変換をどれでもよいから実行し，左再帰的な規則の有無を検査し，これから述べるような形式で，この方法に対する各種の表を生成する自動作成プログラムを書くのは，かなり簡単な練習問題であることを言っておくべきであろう．これも読者に任せることにしよう．

文法を表現するのに構文グラフと呼ばれるリスト構造を用いる．この構造の各節点が，右辺の記号 S を表し，つぎのような 4 個の成分 NAME，DEFINITION (DEF)，ALTERNATE (ALT)，SUCCESSOR (SUC) から成っている．

1. NAME は記号 S そのものの内部表現である．
2. DEFINITION は，S が終端記号のとき 0，非終端記号のときは S に対する規則の最初の右辺の最初の記号に対応する節点へのポインタである．
3. ALTERNATE は，この節点のある右辺のつぎにつづく右辺の最初の記号へのポインタである（つぎの右辺がなければ 0）．これは，この節点が右辺の最初の記号の場合に限る．
4. SUCCESSOR は，右辺内でのつぎの記号へのポインタである（つぎがなければ 0）．

さらに，各非終端記号は，その規則の最初の右辺を指す成分だけを有する節点で表されている．例として，図 4.4 に文法 (4.2.4) の構文グラフを示した．それぞれの節点の各成分の配置はつぎのとおりである．

<table>
<tr><td colspan="3">NAME</td></tr>
<tr><td>DEF</td><td>ALT</td><td>SUC</td></tr>
</table>

(4.2.4)　$E ::= E \langle aop \rangle T \mid T \qquad \langle aop \rangle ::= + \mid -$
$T ::= T \langle mop \rangle F \mid F \qquad \langle mop \rangle ::= * \mid /$
$F ::= (E) \mid I$

この構文グラフは，変換規則 (4.2.2) と (4.2.3) を適用するのに都合の良い表現法である．図4.5 が文法 (4.2.4) の変換後の構文グラフである．この図では，繰返しを示すために節点中にさらに特別な記号を加えている．節点中に ST が記入してあると，この節点の記号から始まる記号列を囲む中かっこが開いて＊印が記入してある節点でそのかっこが閉じる．

ここで注意を一つ．$\{\{XY\}Z\}$ のような右辺に対しては構文解析部の作り方に依っては，X の節点に ST² を記入して X が入れ子になった二つの繰返しの両方の先頭であることを示すようにしなければならないかもしれない．

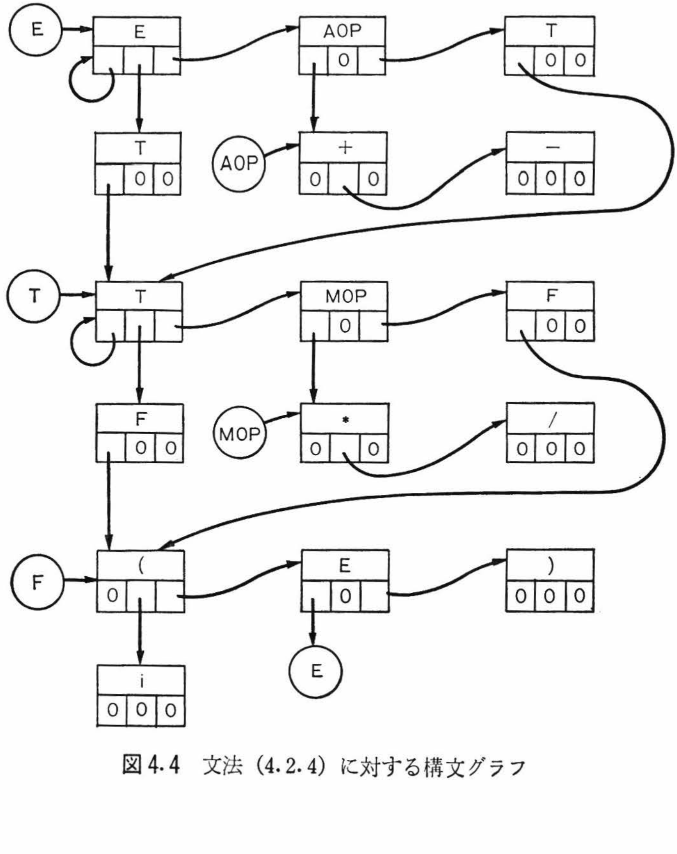


図4.4　文法 (4.2.4) に対する構文グラフ

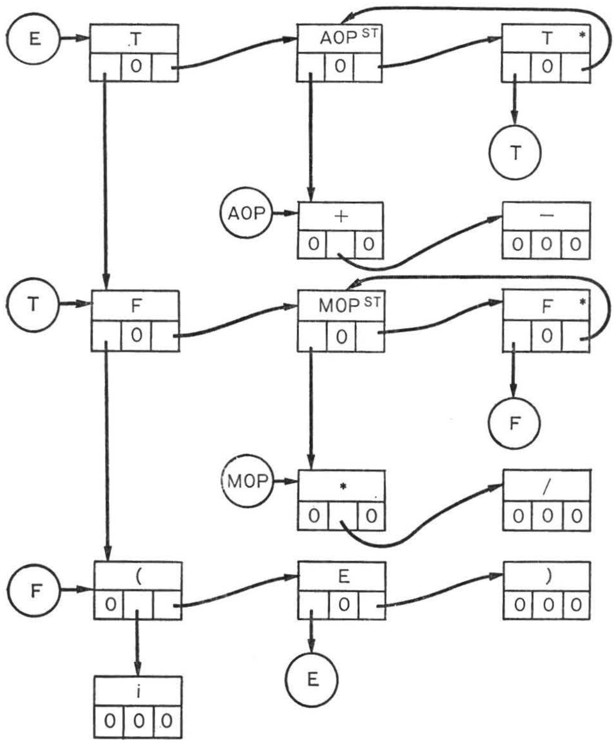


図 4.5 変換後の構文グラフ

また，グラフ上で全く同じ成分からなる節点を同一視することもできよう．例えば，図 4.4 の **NAME** が **F** である 2 個の節点は内容が同じである．これを同一の節点にしても，アルゴリズムは正しく働くが，意味の観点から，これは，良いやり方ではない（これは，後で明らかになる）．

ポインタ変数が使えないような言語でプログラムしなければならない場合には，1 次元配列を用いて，ポインタを配列の要素の指標（添字）で代用することができる．

逆戻りのない構文解析

コンパイラの構文解析部ではなるべく逆戻りをしたくない．我々は，予測した目標が確かに全て正しいものであると信じたい．何となれば，構文と意味を結び付けて，目標を予測し発見しながら問題となっている記号を意味的に処理

してゆくからである．この“処理”には，(1)宣言の処理の際，名前を記号表にのせること，(2)算術式の処理中に，被演算子の型の一致を調べることなどの例がある．

まちがった目標を予測したというので逆戻りをしなければならないとすると，その目標に向っている間に行ってきた意味的処理を，みな元に戻して，なかったことにしてやらねばならない．これは容易なことではない．そこで逆戻りのない構文解析を試みよう．

コンパイラの中で逆戻りをしないようにするために普通用いる文脈は，原始プログラム中でつぎに“カバーされていない”記号である．これができるためには，文法中に選択枝 $x|y|\cdots|z$ がある時はいつでもその文法の $x, y, \cdots, z$ に対する語† を開始する記号の集合が互に共通部分を持たないようになっていなければならない．すなわち，$x \overset{*}{\Rightarrow} Au$ かつ $y \overset{*}{\Rightarrow} Bv$であれば $A \neq B$ である．この要請が満たされれば，$x, y, \cdots, z$ のどれが目標であるかは極めて簡単に決定することができる．ここではかっこで括ることが実際上有効であることを注意しよう．もし $U ::= xy|xz$ であれば，これを $U ::= x(y|z)$ と変換することによって選択枝に対する最初の記号の集合が共通部分を持たないようにするのに役立つ．

ここではこの技法をこれ以上追求するのはやめよう．もっとも次節でこの方法が使われているのがわかるだろう．

4.3　再帰的下向き法

コンパイラの中には，構文解析部が，各非終端記号Uに対して，それぞれ一つずつ再帰的手続きを持っていて，これがUに対する句の構文解析を行うというものがいくつかある．この手続きは，プログラム中でUの句を調べ始めるところで呼出される．したがってこれは，ゴール向きであり予測的である．すなわちそのような句を見出すことができると予測する．この手続きは，プログラムの指定された点とUに対する規則の右辺とを比較して，Uの句を見出し，必要があれば，その下の目標を認識するためにさらに別の手続きを呼出す．

実際，この構文解析の間に 4.1 節の構文解析アルゴリズム（逆戻りなし）で示したのと同じように木が形成される．違っているのは，構文解析を表す記法だけにすぎない．

---

† **訳注**　語は (2.3.6) で定義されている．

この技法を説明するために，下記の文法の非終端記号に対する手続きを書いてみよう：

```
          〈実行文〉 ::= 〈変数〉:= 〈式〉
                     | IF 〈式〉 THEN 〈実行文〉
                     | IF 〈式〉 THEN 〈実行文〉 ELSE 〈実行文〉
(4.3.1)   〈変数〉   ::= i | i ( 〈式〉 )
          〈式〉     ::= 〈項〉 | 〈式〉 + 〈項〉
          〈項〉     ::= 〈因子〉 | 〈項〉 * 〈因子〉
          〈因子〉   ::= 〈変数〉 | ( 〈式〉 )
```

この文法を我々の目的に合せてつぎのように書換える．

```
          〈実行文〉 ::= 〈変数) := 〈式〉
                     | IF 〈式〉 THEN 〈実行文〉 [ELSE 〈実行文〉]
(4.3.2)   〈変数〉   ::= i [( 〈式〉 )]
          〈式〉     ::= 〈項〉 {+ 〈項〉}
          〈項〉     ::= 〈因子〉 {* 〈因子〉}
          〈因子〉   ::= 〈変数〉 | ( 〈式〉 )
```

ここで，逆戻りなしに構文の解析ができるものと仮定する．逆戻りを除去するために用いられる文脈は，すでに構文解析済みの部分的な句に続く，一つの記号だけである．以下に挙げるような約束を行って，擬 ALGOL で，諸手続きを記述しよう．

1. 大域的な変数 NXTSYMB には，原始プログラム中のつぎに処理すべき記号がつねに入っている．新しい目標を見出すために，適当な手続きが呼ばれると，この手続きが最初に調べるべき記号は，すでに，NXTSYMB の中に入っている．
2. 同様に，手続きから戻って，成功を報告する直前には，構文解析を済ませた部分記号列の直後の記号が NXTSYMB の中に入っている．
3. 手続き SCAN は，原始プログラムのつぎの記号を走査してそれを NXTSYMB に入れる．
4. 手続き ERROR は，誤りが検出されたときに呼出される．このときメッセージを印刷して戻ってくる．その後は，あたかも誤りがなかったかの

ごとく処理を続ける（以下の手続きの後にある議論を参照）．

5. 一つの実行文の構文解析を始める前に，SCAN を呼出して，最初の記号を NXTSYMB に入れてから，手続き STATE を呼出す．

```
PROCEDURE STATE;
 IF NXTSYMB = "IF"
 THEN
   BEGIN SCAN; EXPR;
     IF NXTSYMB ≠ "THEN"
     THEN ERROR
     ELSE
       BEGIN SCAN; STATE;
         IF NXTSYMB = "ELSE" THEN
           BEGIN SCAN; STATE END
       END
   END
 ELSE
   BEGIN VAR;
     IF NXTSYMB ≠ ":="
     THEN ERROR
     ELSE BEGIN SCAN; EXPR
            END
   END;
```

〈実行文〉に対するルーチン．ここで最初の記号から実行文の種類を知ることができると仮定するのが普通である．

式の最初の記号を取出して，式に対する手続きを呼出す．引続く記号は，THEN であり，そのつぎは，実行文でなければならない．

ここで再帰的呼出しになる．つぎの記号が ELSE ならばそれに続く実行文を解析する．

IF 文でなければ，代入文でなければならない．変数の構文解析を行い，:= があることを確かめ，

それに続く式を解析する．

実行文に対する手続きの終り．

```
PROCEDURE VAR;
 IF NXTSYMB ≠ "i" THEN ERROR
 ELSE BEGIN SCAN;
     IF NXTSYMB = "(" THEN
       BEGIN SCAN; EXPR;
         IF NXTSYMB ≠ ")"
         THEN ERROR
         ELSE SCAN
       END
   END;
```

〈変数〉に対するルーチン．

まず i から始まらなければならない．

i のつぎの記号を取出す．これが開きかっこでありそのときにかぎり，式の構文解析を行い，その後に閉じかっこがあることを確かめる．

NXTSYMB につぎの記号を入れておく．

```
PROCEDURE EXPR;
 BEGIN TERM;
     WHILE NXTSYMB = "+" DO
       BEGIN SCAN; TERM END
END;
```

〈式〉に対するルーチン．

〈項〉から始まらなければならない．

このあと任意個の "+〈項〉" が繰返される．NXTSYMB が＋でなくなれば終了．

PROCEDURE TERM;	〈項〉に対するルーチン．
BEGIN FACTOR;	このルーチンは〈式〉に対するものと同様
WHILE NXTSYMB = "*" DO	なので説明は不要．
BEGIN SCAN; FACTOR END	
END;	

PROCEDURE FACTOR	〈因子〉に対するルーチン．
IF NXTSYMB = "(" THEN	最初の記号で二つの右辺のどちらかを調
BEGIN SCAN; EXPR;	べる．かっこで括された式の処理．
IF NXTSYMB ≠ ")"	終りが閉じかっこになっていることを確
THEN ERROR	認．
ELSE SCAN	つぎの記号を走査．
END	
ELSE VAR;	変数の場合．

これらの手続きは奇妙なことに，いずれも局所的変数を1個も必要としていない．実際，使われている変数は NXTSYMB ただ1個である．これは，まさしく 4.1 節で用いたスタックを真似るための手続きが実行時に組込まれて，通常のスタック機構が働らいているからにほかならない．

この方法の利点はかなり明白であろう．コンパイラの書き手が，プログラムしながら，手続きの必要に合わせて規則を並べ直すことができる．コンパイラの書き手は，原始言語を良く知っていると仮定しているので，逆戻りが必要ないように自分で規則やプログラムを並べ直すことができる．この方法は，意味処理に対しても非常に融通性に富んでいる．意味処理のための命令を，上の手続きのどこにでも挿入できる．ただし句がちょうど検出された直後は除く．これについては，意味処理を論じるところで詳しく触れたい．

この方法の主な欠点は，部分的に自動化されたシステムの場合よりもプログラミングとデバッギングに手間がかかることである．それでもなお，この方法は，わかりやすく，多くのコンパイラで用いられている．

多くのコンパイラでは，誤りが検出されると実行文の水準までの道すじを全て誤りとしてしまう．この水準では，原始プログラムを BEGIN または END あるいはセミコロンが現れるまで読みとばして，そこから新しく始める．もちろん，これはあまり念の入った方法ではない．この話題は 15 章で採り上げよう．

ちょっと考えれば，適当な文法を受理して，その文法に対する再帰的手続き

群（適当な言語で書かれている）を自動的に生成するようなプログラムが作れそうに思われるだろう．この文法には，当然何らかの制限が課せられよう．

演習問題　4.3

1. 下記の実行文の構文解析をするために，上に挙げた手続きを机上で追って見よ．(a)　`i := i`　(b)　`i := i(i)`　(c)　`IF i THEN i := i.` プログラムは，最初に `SCAN; STATE;` の順で始めよ．

4.4　参考文献概説

言語を記述する各種の表を持った構文解析アルゴリズムを論じた初期の論文には，Irons (61 a) のものがある．彼の方法は，この章で説明したような純然たる下向き解析法ではなくて，下向きと上向き解析法を混合したものであった．上向き法については次章で述べることにしている．Kuno と Oettinger は自然言語の研究にこの下向き解析法を使用した．4.1 節で述べたアルゴリズムは Floyd (64 b) の概説論文から採ったものである．下向き解析法の考え方を父一息子の仕事で説明する方法もこの論文から借用した．コンパイラに予測的な方法を用いる話題に関する指導書となるのは Cheatham と Sattley (64) の SJCC† の論文である．Grau (61) も再帰的手続きを用いたコンパイラを記述している．

逆戻りを禁止する技法は Unger (68) にあり，Lewis と Stearns (68), Rosenkrantz と Stearns (69) は $LL(k)$ 文法と言語を研究している．これは，解析の各段で，これまでに処理した全ての記号とさらに右側の k 個の記号とを調べることによって逆戻りなしに下向き構文解析が可能なものである．

Reynolds (65) の記号処理言語 COGENT は，可能な選択枝を並行して処理し，完全に失敗になったものは削除する．したがって，あいまいな言語も取扱うことができる．すなわち，あいまいな文に対する構文の木が全て生成される．下向き解析は，META コンパイラ作成システム (Schorre (64)) が得意とする問題であり，最も古いコンパイラコンパイラ――Brooker ほか (63)――で用いられている．これは Rosen (64) に載っている．再帰的下向き法を用いたコンパイラには，Burroughs の拡張 ALGOL や SHARE 7090 ALGOL コンパイラなどがある．

---

† **訳注**　SJCC：参考文献のまえがきの略号参照のこと (540 ページ)．

第 5 章

単 純 順 位 文 法

5.1 順位関係とその利用

本章では，上向き法を用いて構文を解析する技法を説明する．上向き法とは，規則 $U ::= u$ を用いて，現在の文形式から把手 u（最も左側の単純句）を見つけてそれを非終端記号 U に還元することを繰返しつつ構文解析を行うものであることを思い出そう．上向き解析法に伴う問題点は，把手を見出しそれをどの非終端記号に還元するかを知ることにある．本章では，この問題の解決を（単純）**順位文法** (precedence grammar) と呼ばれるクラスの文法について示す．これは，もちろん，全ての文法に対する答えとはなっていない．ここでは，つねに，左から右に，下から上に進むので，ここで現れる文形式は全て正準であり，生成される導出も全て正準である．したがって，本章では，"文形式" と書けば，それは "正準な文形式" を意味することにしよう．

文形式 x が与えられると，そこからどのようにして把手を見出すことができるのか．左から右に進んで，2個の記号を同時に見ながら，把手の**後部** (tail) を見付けたらそれを知らせるようにしたい．そして，この文形式の左端に向って戻り，再び同時に2個の記号だけを用いて把手の頭部を見付けて，決定を下したいのである．すなわち，つぎのような問題に直面している．記号列 $\cdots RS\cdots$ が与えられたとき，R がつねに把手の後部であるのか，RS がともに把手の中に現れることがあるのか，それ以外か．とにかく構文解析を始める前に文法を調べ，2個の記号の全ての組合せについて，決定表を作っておきたい．

文法 G の語彙に含まれる2個の記号 R と S を考えよう．（正準な）文形式 $\cdots RS\cdots$ があるとすると，正準な構文解析の適当な時点で R または S

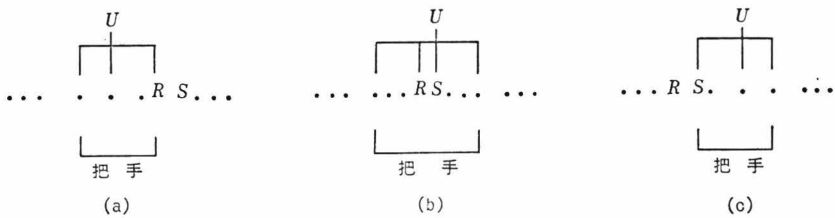


図5.1　順位関係の説明図

(あるいは両者とも) が把手の中に含まれるにちがいない．つぎのような三つの可能性が生ずる．

1. R は把手の一部であり，S は一部ではない (図5.1**a**)．この場合 $R \gtrdot S$ と書き，R は S より大，あるいは R は S より**順位が上**という．これは，R がまず還元されるからである．R は，適当な規則 $U ::= \cdots R$ の最後の記号であることに注意しよう．把手は S の左側に位置しているから，S は終端記号でなければならないことに注意．
2. R と S が共に把手に含まれる (図5.1**b**)．これを $R \doteq S$ と書く．両者は同じ順位を持ち，同時に還元されなければならない．明らかに，文法には，$U ::= \cdots RS \cdots$ という規則が存在しなければならない．
3. S は把手の一部であり，R が一部でない (図5.1**c**)．これを $R \lessdot S$ と書き R は S より小と読む．S は $U ::= S \cdots$ という規則の頭部でなければならない．

$\cdots RS \cdots$ という形式の (正準な) 文形式が存在しなければ，順序対 (R, S) には，何の関係もないという．新しい三つの関係 $\lessdot$，$\doteq$，$\gtrdot$ は，対称ではない．例えば，$S \lessdot R$ から $R \lessdot S$ はでてこない．

例として，つぎのような文法 $G5[Z]$ を考えよう：

$$
\begin{aligned}
(5.1.1)\quad Z &::= b \;\; M \;\; b \\
M &::= (\;\; L \;\; | \;\; a \\
L &::= M \;\; a \;\;)
\end{aligned}
$$

言語 $L(G5)$ は，bab，$b(aa)b$，$b((aa)a)b$，$b(((aa)a)a)b$ などの記号列を

含んでいる．下記の表は，文形式，構文の木，その木の把手とその木から導出される関係を対応づけて示したものである．

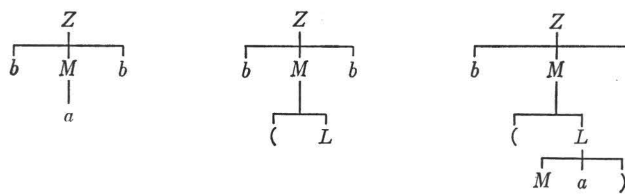
文形式	$b\ a\ b$	$b\ (\ L\ b$	$b\ (\ M\ a\ )\ b$
構文の木			
把　手	a	$(\ L$	$M\ a\ )$
木から得られる関係	$b \lessdot a$ $a \gtrdot b$	$b \lessdot ($ $(\doteq L$ $L \gtrdot b$	$(\lessdot M$ $M \doteq a$ $a \doteq)$ $) \gtrdot b$

図5.2に文法 $G5$ の順位行列（precedence matrix）——全ての順位関係を示す行列——を与えた．この行列の要素 $B[i, j]$ は，記号の対 (S_i, S_j) に対する関係を述べている．空白は，二つの記号の間に関係が存在しないことを示している．

最初は，文法の記号に対する全ての関係を見出すのは難しく思うかもしれない．全ての関係を見出すには，構文の木をたっぷり調べなければならないように思える．全ての関係を見出したことを，どうして確かめるのか．次節で，任

	Z	b	M	L	a	(	)
Z							
b			$\doteq$		$\lessdot$	$\lessdot$	
M		$\doteq$			$\doteq$		
L		$\gtrdot$			$\gtrdot$		
a		$\gtrdot$			$\gtrdot$		$\doteq$
(			$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	
)		$\gtrdot$			$\gtrdot$		

図5.2　文法 $G5$ に対する順位行列

意の文法に対して，これらの関係を極めて容易に構成できる方法を用いて，上の関係を定義しなおそう．

これらの関係は，われわれが構文解析をするのにどのように役立っているのであろうか．もし，任意の記号対 (R, S) の間に複数の関係が存在すれば，その関係は，全く役に立たなくなる．しかしながら，二つの記号の間に成立つ関係が高々1個であれば，この関係が，任意の文形式の把手を示してくれる．この場合，任意の文形式 $S_1\cdots S_n$ の把手は，つぎのような関係

$$(5.1.2)\quad \begin{aligned} &S_{j-1} \lessdot S_j \\ &S_j \doteq S_{j+1} \doteq S_{j+2} \doteq \cdots \doteq S_i \\ &S_i \gtrdot S_{i+1} \end{aligned}$$

を満たす最も左側の部分記号列 $S_j\cdots S_i$ であるといえる（この表明は，把手の記号 S_j (S_i) が文形式の最初の（最後の）記号である場合をも取扱うには幾分変更を必要とする．これは後述しよう）．把手 $S_j\cdots S_i$ が (5.1.2) を満たしていることは，関係の定義から，明白である．しかし，(5.1.2) を満たす最も左側の部分記号列が把手であるということはさほど明らかなことではない．この表明は，次節で証明しよう．ここではこれを証明なしに認めて，図5.2 の順位行列を用いて文法 $G5$ の $b(aa)b$ という文を構文解析する例題を解いてこの節を終ることにしよう．図5.3 が，構文解析列である．ステップごとに，文形式と記号間に存在する関係を図5.2 に従って示している．

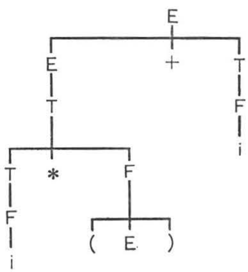
ステップ	文形式	把手	還元された結果	構成される直接導出
1	$b \lessdot (\lessdot a \gtrdot a \doteq) \gtrdot b$	a	M	$b(Ma)b \Rightarrow b(aa)b$
2	$b \lessdot (\lessdot M \doteq a \doteq) \gtrdot b$	$Ma)$	L	$b(Lb \Rightarrow b(Ma)b$
3	$b \lessdot (\doteq L \gtrdot b$	$(L$	M	$bMb \Rightarrow b(Lb$
4	$b \doteq M \doteq b$	bMb	Z	$Z \Rightarrow bMb$

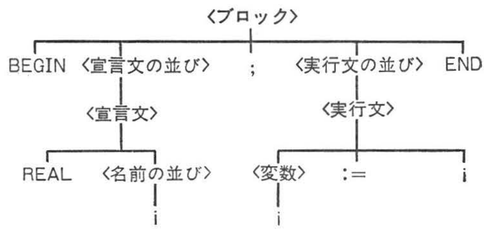
得られた導出：$Z \Rightarrow bMb \Rightarrow b(Lb \Rightarrow b(Ma)b \Rightarrow b(aa)b$

図5.3　文形式 $b(aa)b$ の構文解析

演習問題 5.1

1. 下記の構文の木から全ての順位関係を決定せよ（把手から関係を決定し終ったら，その把手は木から取外し次の木を作る．新しい木からまた関係を決定することができる）．





2. 文法 $G5$ の以下の文から，図 5.2 の順位行列を用いて把手を見出し構文解析せよ：bab, $b(aa)b$, $b(((aa)a)a)b$.

3. 文法 (4.1.1) の順位関係を全て見出せ．

5.2 関係の定義と構成

前節では，文形式の構文の木を用いて三つの順位関係を定義した．本節では，これを文法の規則だけを用いて定義し直したい．そうすれば，関係の構成法がわかって，前節の表明を証明することができる．これらの定義は，一見手強そうであるが，われわれの表明を証明するのには是非とも必要である．

(5.2.1) **定義**　文法 G に対して，この語彙 V の記号対の間の**順位関係**を以下のように定義する．

1. $R \doteq S$ とは，G に規則 $U ::= \cdots RS \cdots$ が存在しそれに限られることをいう．
2. $R \lessdot S$ とは，規則 $U ::= \cdots RV \cdots$ が存在し，関係 V FIRST$^+$ S ((2.6.4) 参照) が成立ちそれに限られることをいう．
3. $R \gtrdot S$ とは，S が終端記号でかつ規則 $U ::= \cdots VW \cdots$ が存在し，関係 V LAST$^+$ R と W FIRST* S ((2.6.7) および (2.6.4) 参照) が成立ちそれに限られることをいう．■

つぎのような二つの関係もしばしば必要になる．

$R \leqq S$ とは，$R \doteq S$ または $R \lessdot S$ が成立つことをいう．
$R \geqq S$ とは，$R \doteq S$ または $R \gtrdot S$ が成立つことをいう．

このように慣れない方法で定義された上の関係が 5.1 節のもっとわかりやすい定義と同等であることを証明しなければならない．ここでは，$\doteq$ と $\gtrdot$ について証明を行う．$\lessdot$ に関する証明は読者に任せよう（演習問題 1）．

(5.2.2)　**定理**　ある文形式の把手の中に部分記号列として RS が現れるならば，そしてそのときに限り $R \doteq S$ が成立つ．

証明　$R \doteq S$ と仮定する．把手の中に RS を持った構文の木を構成しなければならない．定義により，適当な u, v に対して，規則 $U ::= uRSv$ が存在する．文法が既約であると仮定すると，全ての規則は，少くとも一つの文の導出に用いられるから，$Z \overset{*}{\underset{c}{\Rightarrow}} xUy$ が x, y に対して存在する．ここで Z は出発記号である．求める構文の木をつぎの3段階で構成する．

1. xUy の導出に対して木を作る．
2. ステップ1. で構成された木を還元して（把手を取払って），U が把手の一部となるまで繰返す．これは，全ての記号が，いつかは把手の一部になるので可能である．
3. ステップ2. で得られた木に規則 $U ::= uRSv$ に応じて枝を追加する．U が把手の中に含まれていたので，新しい把手は $uRSv$ であり，これによって求める木が得られた．

逆に，もし RS が把手の一部であれば，把手の定義から，$U ::= \cdots RS \cdots$ という規則が存在しなければならない．■

(5.2.3)　**定理**　R が把手の最後の記号であるような正準な文形式が存在するならば，そしてそのときに限り $R \gtrdot S$ が成立つ．

証明　$R \gtrdot S$ と仮定する．要求された属性をもった構文の木を構成する方法を示そう．まず，$R \gtrdot S$ だから，規則 $U ::= uVWv$ が存在して，ここで V LAST$^+$ R かつ W FIRST* S である．この意味するところは，文法は既約であるから，$Z \overset{*}{\Rightarrow} xUy \Rightarrow xuVWvy$ である（Z は出発記号）．つぎに，V LAST$^+$ R は $V \overset{+}{\Rightarrow} \cdots R$ を意味し，これから

(5.2.4) $Z \stackrel{+}{\Rightarrow} xuVWvy \stackrel{+}{\Rightarrow} xu\cdots RWvy$

が得られる．求める木はつぎのようにして構成する．

1. 導出 (5.2.4) に対する構文の木を描く．
2. 把手をつぎつぎと還元していって R が把手の一部となるまで繰返す．$V \stackrel{+}{\Rightarrow} \cdots R$ であるから，R は把手の最後の記号でなければならないことに注意．
3. 定義により W FIRST* S が成立つから，$W \stackrel{*}{\Rightarrow} S\cdots$ である．ステップ 2. で得られた構文の木に，この導出に対する枝を加える．明らかに，ここでは，把手はそのままである．何となれば，把手は R を含み，新しい枝は R の右側に位置しているからである．したがって，R は把手の最後の記号であり，そのつぎに S が続く．

逆に，R が把手の最後の記号で，つぎに S が続くような木を仮定しよう．まず，正準な構文解析となる還元を続けて S が把手に含まれるまで繰返す．これによって，文形式 $\cdots VS\cdots$ に対する木が，$V \stackrel{+}{\Rightarrow} \cdots R$ となるような適当な V に対して生じる．二つの場合が生じうる．

1. もし V と S が把手に含まれるならば，終了．何となれば，U ::= $\cdots VS\cdots$ という規則が存在しなければならず，U ::= $\cdots VW\cdots$ が得られるからである．ただし $V \stackrel{+}{\Rightarrow} \cdots R$ かつ $W = S$.
2. もし S が把手の頭部であれば，還元を続けて V がついには把手の中に入るような適当な V に対して文形式 $\cdots VW\cdots$ が得られるまで繰返す．$W \stackrel{+}{\Rightarrow} S\cdots$ に注意．さらに，W も把手に含まれなければならない．何となれば，もし V が把手の最後の記号ならば，それは，最初から記号列の把手の最後の記号でもあることになり，S が把手に含まれるという前提に矛盾する．

かくて，$V \stackrel{+}{\Rightarrow} \cdots R$ かつ $W \stackrel{*}{\Rightarrow} S\cdots$ となる規則 U ::= $\cdots VW\cdots$ が存在する．証明おわり■

(5.2.5) **定理** 把手の頭部として S を持つような文形式 $\cdots RS\cdots$ が存在するならば，そしてそのときに限り $R \lessdot S$ が成立つ．

証明 証明は読者に任せる (演習問題 1).■

以上で，形式的な定義 (5.2.1) が，5.1 節で与えた構文の木による形式ば

らない定義と同等であることがわかった．読者にとっては，構文の木で与えた定義を勉強することが重要である．完全に理解してから，先に進んでほしい．これで，単純順位文法を定義する用意ができた．

(5.2.6) **定義**　文法 G は，以下の仮定が成立てば，（単純）**順位文法** (precedence grammar) または **(1, 1) 順位文法** と呼ばれる．

1. この文法の語彙に含まれる任意の二つの記号の間に高々1個の関係が成立つ．
2. どのような二つの生成規則も同じ右辺を持つことはない．■

(1, 1) は，把手になりうる句の両端に，それが把手か否かを決定するのに役立つ記号が1個づつ用いられている事実に由来する．すなわち，もし記号 R が把手に含まれていれば，R がその把手の後部であるか否かを知るためには，R のつぎの記号 S だけを見ればよい．頭部の場合も同様である．もし G が順位文法ならば，この関係が，任意の文形式の把手を見出す方法を与えてくれて，上の第二の条件によって，この把手が唯一の非終端記号に還元できることが保証される．これは，以下の定理で証明しよう．定理の前に，もう一つ議論しておかなければならない問題がある．把手というのは，以下に示す (5.2.9) を満足する最も左側の記号列 $S_j \cdots S_i$ であることを示したい．しかしながら，文形式の最初の記号が，把手に含まれる場合には注意しなければならない．何となれば，この場合，$S_0 \lessdot S_1$ となるような S_0 が存在しないからである．この問題は，

(5.2.7)　文形式は全て # と # で括られる（ここで # は文法で使用されない記号とする）．さらに，文法の任意の記号 S に対して，$\# \lessdot S$ および $S \gtrdot \#$ が成立つとしておく．■

と仮定することで解決しよう．この仮定のもとに，つぎの定理を証明する．

(5.2.8) **定理**　順位文法はあいまいではない．さらに，任意の文形式 $S_1 \cdots S_n$ の唯一つの把手は，

(5.2.9)　$S_{j-1} \lessdot S_j \doteq S_{j+1} \doteq \cdots \doteq S_i \gtrdot S_{i+1}$

を満たすような，最も左側の部分記号列 $S_j \cdots S_i$ である．

証明　まず任意の文形式の把手 x が一意的であることを示そう．もし

$S_k \cdots S_p$ が把手であれば，明らかに

$$S_{k-1} \lessdot S_k \doteq \cdots \doteq S_p \gtrdot S_{p+1}$$

である．これは，$\lessdot$, $\doteq$, $\gtrdot$ の定義から直ちに導かれる．把手として (5.2.9) を満たす，最も左側の部分記号列 $S_j \cdots S_i$ を持たないような構文の木が存在すると仮定しよう．すると，何回枝を取除くステップを行っても，この部分記号列を把手にすることにならない．何となれば，もしいつかこの部分記号列が，最も左側の完全な枝になるのならば，これが今もそうなるからである．

正準な構文解析を続けると，S_{j-1}, S_j, $\cdots$, S_{i+1} のどの記号もいつかは把手の一部に現れなければならない．そのような最初の記号を S_t とする．その把手は $S_j \cdots S_i$ でありえないので，つぎのいずれかが成立たねばならない．

1. $t = j-1$: $S_{j-1} \doteq S_j$ または $S_{j-1} \gtrdot S_j$ のとき；
2. $t = i+1$: $S_i \doteq S_{i+1}$ または $S_i \lessdot S_{i+1}$ のとき；
3. $j \leq t \leq i$:
   a) $S_{j-1} \doteq S_j$ で，S_{j-1} が把手に含まれる
   b) $S_i \doteq S_{i+1}$ で，S_{i+1} が把手に含まれる
   c) $S_{k-1} \lessdot S_k$ で，S_k が把手の頭部になるような $j < k \leq t$ を満たす k が存在する
   d) $S_k \gtrdot S_{k+1}$ で，S_k が把手の後部になるような，$t \leq k < i$ を満たす k が存在する．

いずれの場合も，S_{j-1}, $\cdots$, S_{i+1} 中の 2 個の記号の間に複数個の関係が存在して順位文法の定義にもとることがわかる．よって，$S_j \cdots S_i$ は一意的な把手でなければならない．

かくて，任意の文形式 x の構文解析の各段で，把手が一意的に決定される．この把手を右辺とする規則は唯一つなので ((5.2.6) 条件 2)，この把手を還元しうる記号は唯一つ存在する．したがって，構文解析の各段の直接還元は一意的に決まる．これは，任意の文に対する構文の木が一意的であり，文法があいまいでないことを意味している．■

順位文法の文の構文解析は，諸関係が一たん決ると容易なことがわかる．この文法が，おそらく，実用に供されるもののうちで，最も簡単なクラスのものであろう．これらの順位関係自体，実際の定義が多少込入っていたとしても，5.1 節でみたように，理解するのはかなり容易である．後になってから，定義

はより面倒であるがより実用的な文法のクラスを取扱うことにして，しばらく，この順位文法を少し扱って，順位関係を作成するアルゴリズムや，順位文法の文に対する実際の認識法——構文解析部——のアルゴリズムを与えよう．

順位関係の構成

順位関係$\doteq$の構成は簡単であり，説明の必要はあるまい．規則の右辺をたどって $\cdots RS\cdots$ となっているような R と S に対して $R\doteq S$ と設定するだけでよい．

順位関係 $\lessdot$ と $\gtrdot$ は，もう少し作業が必要であるが，計算はやはり極めて単純である．2.7 節のブール行列の理論を適用するだけでよい．

(5.2.10)　**定理**　順位関係 $\lessdot$ は，関係 $\doteq$ と FIRST^+ の積に等しい．すなわち

$$\lessdot \;=\; (\doteq)(\mathrm{FIRST}^+)$$

証明　これは，関係の積の定義と (5.2.1) の $\lessdot$ の定義から直ちに導びかれる．■

(5.2.11)　**定理**　I を恒等関係とすると，S が終端記号でかつ

$$R\,(\mathrm{TRANSPOSE}\,(\mathrm{LAST}^+))\,(\doteq)\,(\mathrm{I}+\mathrm{FIRST}^+)\;S$$

が成立ちかつそのときに限り $R\gtrdot S$ が成立つ．

証明　証明は読者に任せる（演習問題 4)．■

定理 (5.2.10) は，順位関係 $\lessdot$ が，(1) 関係 FIRST に対するブール行列 F を構成し，(2) いわゆる，Warshall のアルゴリズムを用いて行列 F^+ を作り，(3) 順位関係に対するブール行列 EQ を作り，(4) EQ と F^+ を乗ずることによって得られることを示している．順位関係 $\gtrdot$ の計算も同様に簡単である．

演習問題　5.2

1. S が把手の頭部であるような文形式 $\cdots RS\cdots$ が存在しかつそのときに限り $R\lessdot S$ であることを証明せよ．
2. 下記の文法は順位文法ではないことを示せ．

$$Z ::= bEb \qquad E ::= E+T|T$$

3. 下記の文法は順位文法ではないことを示せ.

$$Z ::= b\,E1\,b,\ \ E1 ::= E|E+T|T|i,\ \ T ::= i|(E1)$$

4. 定理 (5.2.11) を証明せよ.
5. 定理 (5.2.10) と (5.2.11) を用いて, 文法 (5.1.1) の順位関係を構成せよ.
6. 順位関係を構成して印刷するプログラムを適当な高水準言語で書きデバッグせよ. このプログラムの入力は, 適当な形式で記述した文法である.

5.3 構文解析アルゴリズム

文の認識のために順位関係を実際に用いる時には, この関係を表すためのコンパクトな方法が欲しい. つぎのような符号化された値をもった行列 P を用いる技法が普通である.

$P[i, j]=0$　S_i と S_j の間に関係がないとき
$P[i, j]=1$　$S_i \lessdot S_j$ のとき
$P[i, j]=2$　$S_i \doteq S_j$ のとき
$P[i, j]=3$　$S_i \gtrdot S_j$ のとき

この技法が使えるのは, 順位文法では, 任意の2個の記号の間に高々1個の関係しかないことがわかっているからにほかならない.

規則の方の構造は, 右辺が与えられたら, その右辺を規則の中から見付け出して, それに対応する左辺が取出しやすいようにして表に納めておかなければならない. 構文解析アルゴリズムは, 以下のように働く. 入力記号列の記号は左から右に処理されてスタック S に納める. これをスタックの最上段にある記号とつぎに入ってくる記号の間に順位関係が見付かるまで続ける. この関係が見付かったことは, スタックの最上段の記号が, 把手の後部であり, したがって把手全体がスタック中にあることを意味している. そしてこの把手を規則の表の中から探して, この把手を還元すべき, 対応する非終端記号と, スタックの中で入替える. この処理を繰返してついには, スタックの中には Z (出発記号) だけが残り, つぎの入力記号が # となるまで続行する.

図5.4は, この認識法の流れ図を描いたもので, 図中で用いられている記号は以下のとおりである.

1. S は記号を納めるためのスタック, i がそのカウンタである.

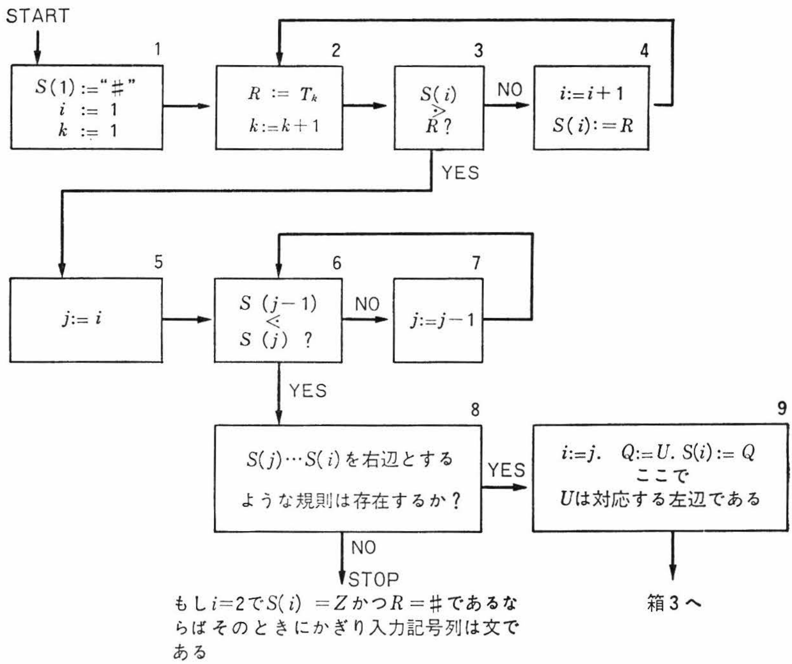


図5.4　単純順位認識法

2. j は，スタックの上から数個の要素を参照するための指標である．
3. $T_1T_2\cdots T_n$ が構文解析すべき文である．まず，スタックの中に文の左端の区切り記号 # を入れる．この記号は T_{n+1} として文の終端にも付加されていると仮定している．
4. Q と R は記号を値として持つ変数で，構文解析の進行につれて，適宜記号 t を代入する．

もし $T_1\cdots T_n$ が実は文ではないことになれば，アルゴリズムはそこで停止し，このことを表示する．流れ図中の番号のついた箱に関する注釈はつぎのとおりである．

箱 1. スタックの最上段に文の左端の区切り記号を置いてスタックを初期設定する．文中の記号への指標 (k) を最初の記号に設定する．

箱 2. つぎの記号をとってくる．これを R に代入し k を1増す．

箱 3. 記号の対 $(S(i), R)$ が $\gtrdot$ の関係になければ，把手はまだスタック中に入りきっていないので，R をスタックに押し込みつぎの記号をとってくる．

箱 5—7. $S(i) \gtrdot R$ が成立した場合．把手はスタック中にあり，この三つの箱で，把手の先頭の記号を探している．

箱 8—9. 記号列 $S(j)\cdots S(i)$ を検査する．もしこの記号列が規則の右辺になければ，アルゴリズムは終了である．もし，$i=2$ で $S(i)=Z$（出発記号）でありかつこのときに限り，入力記号列は文であったことになる．もしこの記号列と合致するものが規則の右辺の中に存在すれば，スタックからこの把手を取除いて（箱9），スタックには代りにこれから還元された非終端記号をのせて，つぎの把手を見付けるために箱3に戻る．

このような認識法では，余程ひねくれた文法でない限り入力記号列全体を，同時に記憶装置の中に置いておく必要がないというのが都合の良い点である．記号は1個ずつ入力装置から読込まれスタックに置かれ，把手が還元されると消滅する．もし把手が，記号列の右端にあれば，記号列全体がスタックに入らなければならないが，われわれのプログラム用言語の文法の構造にはそのようなものはない．

この流れ図を用いて，文法 (5.1.1) の文 $b(aa)b$ をもう一度構文解析してみよう．図5.5 にこの解析の様子を示した．図には，各ステップにおけるス

ステップ	S_1 $S_2\cdots$	関係	R $T_k\cdots$
0	#	$\lessdot$	b (a a) b #
1	# b	$\lessdot$	(a a) b #
2	# b (	$\lessdot$	a a) b #
3	# b (a	$\gtrdot$	a) b #
4	# b (M	$\doteq$	a) b #
5	# b (M a	$\doteq$	) b #
6	# b (M a)	$\gtrdot$	b #
7	# b (L	$\gtrdot$	b #
8	# b M	$\doteq$	b #
9	# b M b	$\gtrdot$	#
10	# Z		#

図5.5 $b(aa)b$ の構文解析

タックの内容，入力されたばかりの記号 R，スタック中の記号 $S(i)$ と R の関係，まだ読込まれていない文形式の残部を，箱3の直前の状態で示してある．

演習問題 5.3

1. 図5.4のアルゴリズムと例5.1.1の文法 $G5$ の順位行列を用いて下記の記号列（必ずしも文とは限らない）の構文解析を試みよ．

   $\#aa\#$, $\#((aa)a)\#$, $\#(\ )\#$, $\#((((aa)a)a)a)\#$

2. 図5.4の構文解析アルゴリズムをプログラムしデバッグせよ．このためには，まず順位行列の形式（これは演習問題5.2.6のプログラムの出力のはずである）を決定しなければならない．つぎにこの構文解析部への入力は，演習問題3.3.2でプログラムしたような走査部の出力にしてよいだろう．いずれにしても，各記号の内部表現は，順位行列の行番号または列番号にすべきであろう．

5.4 順位関数

順位行列は莫大な記憶容量が必要になりがちである．160個の記号を有する言語では，少くとも2ビットの要素からなる 160 × 160 行列が必要である．しかしながら全ての記号に対してこの行列の情報を

(5.4.1) $R \doteq S$ ならば $f(R) = g(S)$
$R \lessdot S$ ならば $f(R) < g(S)$
$R \gtrdot S$ ならば $f(R) > g(S)$

となるような二つの関数 f, g を用いて表現できる場合がかなりある．これを行列の**1次元化** (linearization) と呼んでいる．これによって，$n \times n$ 個の記憶場所を $2 \times n$ 個に縮少することができる．図5.2の順位行列に対する関数 f, g は

(5.4.2)

	Z	b	M	L	a	(	)
f	1	4	7	8	9	2	8
g	1	7	4	2	7	5	9

となる．このような関数は一意的でないことに注意しよう．このような関数は

一つ見つかれば，無限個存在するのである．このような関数が存在しないような順位行列も多い．つぎのような，記号 S_1，S_2 に対する 2×2 順位行列でさえ，1次元化することができない．

$$\begin{bmatrix} \doteq & > \\ \doteq & \doteq \end{bmatrix}$$

何故か．もし関数が存在したとすれば，(5.4.1) から

$$f(S_1) > g(S_2),\quad g(S_2) = f(S_2)$$
$$f(S_2) = g(S_1),\quad g(S_1) = f(S_1)$$

でなければならない．これから $f(S_1) > f(S_1)$ となり矛盾が導かれる．

順位関数が存在する場合にこれを構成する方法はつぎのとおりである．

(5.4.3) **定理** 順位関数を構成するために以下のステップを踏む：

1. $2\times n$ 個の節点を持った有向グラフを描く．節点の名を $f_1, \cdots, f_n$, $g_1, \cdots, g_n$ とする．もし関係 $S_i \geqq S_j$ が存在すれば，f_i から g_j に矢印を描く．もし $S_i \leqq S_j$ ならば g_j から f_i に矢印を描く．
2. 各節点に対して，その節点から到達しうる節点の数（自分も含めて）を番号として割付ける．f_i に割付けられた番号が $f(S_i)$ であり，g_i に割付けられた番号が $g(S_j)$ である．
3. でき上った関数 f, g について，(5.4.1) に従って，もとの関係と矛盾しないかを検査する．もし矛盾がなければ f と g は有効な関数である．もし矛盾があれば，順位関数は存在しえない．

証明 まず $S_i \doteq S_j$ ならば $f(S_j) = g(S_j)$，$S_i < S_j$ ならば $f(S_i) < g(S_j)$，$S_i > S_j$ ならば $f(S_i) > g(S_j)$ を示さねばならない．最初のものは，構成法から直ちに導かれる．何となれば，もし $S_i \doteq S_j$ ならば f_i から g_j へ向かう矢印と g_j から f_i に向う矢印がともに存在し，一方から到達できる節点は他方からも到達できるからである．残りの二つの条件は対称的なので，$S_i > S_j$ ならば $f(S_i) > g(S_j)$ を証明しよう．$S_i > S_j$ だから，f_i から g_j へ向う矢印が存在する．したがって，g_j から到達可能な節点は全て f_i からも到達可能である．すなわち $f(S_i) \geq g(S_j)$．ここで等号がなりたてば $(f(S_i) = g(S_j))$，順位関数が存在しないことだけを云えばよい．

$f(S_i) = g(S_j)$ と仮定しよう．するとつぎのような順路が存在しなければならない．

$$f_i \to g_j \to f_k \to g_l \to \quad \cdots \quad \to g_m \to f_i$$

したがって,

$$S_i \gtrdot S_j,\ S_k \leqq S_j,\ S_k \geqq S_l,\ \cdots,\ S_i \leqq S_m$$

となる．この系列は，任意の関数 f と g に対して，$f(S_i) > f(S_i)$ でなければならないことを示している．これは矛盾である．■

有向グラフは，節点の集合の上のある関係を記述するものである．すなわち節点 x から節点 y へ向かう矢印が存在してかつそのときに限り xRy が成立つ．したがって，$2n \times 2n$ のブール行列 B によってつぎのように構成用の最初のグラフを表すことができる．

(5.4.4)
$$B := \begin{bmatrix} O & GE \\ LE^T & O \end{bmatrix}$$

ここで GE は関係 $\geqq$ に対する $n \times n$ 行列であり，LE^T は，関係 $\leqq$ に対する行列を転置したものである．すなわち，$i, j = 1, \cdots, n$ に対して，

(5.4.5)　$B[i, j] = B[n+i, n+j] = 0$
$B[i, n+j] = 1$ ： $S_i \geqq S_j$ のときに限る（右上の四半分）
$B[n+j, i] = 1$ ： $S_i \leqq S_j$ のときに限る（左下の四半分）

第1行から第 n 行は $f_1, \cdots, f_n$ に対応し，第 $n+1$ 行から第 $2n$ 行は $g_1, \cdots, g_n$ に対応する．つぎの定理の証明は読者に任せよう．

(5.4.6)　**定理**　順位関数が存在すれば，それはつぎの構成法によって，生成できる．

1. (5.4.4) の行列を構成する．
2. B の反射的推移的閉包 B^* を構成する．
3. $B^*[j, k] = 1$ となるような k の個数を $f(S_j)$ の値とする．
   $B[n+j, k] = 1$ となるような k の個数を $g(S_j)$ の値とする．■

図5.2 の順位行列を例にとろう．

$$GE=\begin{bmatrix}0&0&0&0&0&0&0\\0&0&1&0&0&0&0\\0&1&0&0&1&0&0\\0&1&0&0&1&0&0\\0&1&0&0&1&0&1\\0&0&0&1&0&0&0\\0&1&0&0&1&0&0\end{bmatrix}\qquad LE^T=\begin{bmatrix}0&0&0&0&0&0&0\\0&0&1&0&0&0&0\\0&1&0&0&0&1&0\\0&0&0&0&0&1&0\\0&1&1&0&0&1&0\\0&1&0&0&0&1&0\\0&0&0&0&1&0&0\end{bmatrix}$$

だから

$$B=\begin{bmatrix}0&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0&1&0&0&1&0&0\\0&0&0&0&0&0&0&0&1&0&0&1&0&0\\0&0&0&0&0&0&0&0&1&0&0&1&0&1\\0&0&0&0&0&0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0&1&0&0&1&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0&0&0&0&0&0\\0&1&0&0&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0&0&0&0&0&0&0\\0&1&1&0&0&1&0&0&0&0&0&0&0&0\\0&1&0&0&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0&0&0&0&0&0&0\end{bmatrix}$$

$$B^*=\begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&1&0&0&0&1&0&0&0&1&1&0&0&0\\0&1&1&0&0&1&0&0&1&1&1&1&0&0\\0&1&1&1&0&1&0&0&1&1&1&1&0&0\\0&1&1&0&1&1&0&0&1&1&1&1&0&1\\0&0&0&0&0&1&0&0&0&0&1&0&0&0\\0&1&1&0&0&1&1&0&1&1&1&1&0&0\\0&0&0&0&0&0&0&1&0&0&0&0&0&0\\0&1&1&0&0&1&0&0&1&1&1&1&0&0\\0&1&0&0&0&1&0&0&0&1&1&0&0&0\\0&0&0&0&0&1&0&0&0&0&1&0&0&0\\0&1&1&0&0&1&0&0&1&1&1&1&0&0\\0&1&0&0&0&1&0&0&0&1&1&0&1&0\\0&1&1&0&1&1&0&0&1&1&1&1&0&1\end{bmatrix}$$

$i=1, \cdots, n$ に対して B^* の第 i 行の 1 の個数を算えて (5.4.2) の $f(S_i)$ になる．同様に $j=1, \cdots, n$ 対して，第 $n+j$ 行の 1の個数を算えて (5.4.2) の $g(S_j)$ になる．

コンパイラはプログラム中に誤りを発見すると，それを正して，さらに誤りを見付けるために構文解析を続けるべきであろう．誤りを正すためには，できるだけ早くその誤りを見出すことが重要なことがある．先のような関数を用いると誤りの検出を遅らせる可能性がある．1次元化によって2個の記号の間の実際の関係の有無に関する情報がもはや失われてしまうからである．これを説明するために記号列 $b\ a$))))) b の構文解析を文法 $G5$ に従ってまず順位行列を用いて，つぎに順位関数を用いて試みてみよう．

1)　行列を用いた構文解析

ステップ	$S_1\ S_2\cdots$	関係	$R\ T_k\cdots$
0	#	$\lessdot$	b a))))) b #
1	# b	$\lessdot$	a))))) b #
2	# b a	$\doteq$	))))) b #
3	# b a)	なし	)))) b #

2)　関数 f，g を用いた構文解析

ステップ	$S_1\ S_2\cdots$	f 関係 g	$R\ T_k\cdots$
0	#	1 $<$ 7	b a))))) b #
1	# b	4 $<$ 7	a))))) b #
2	# b a	9 $=$ 9	))))) b #
3	# b a)	8 $<$ 9	)))) b #
4	# b a))	8 $<$ 9	))) b #
5	# b a)))	8 $<$ 9	)) b #
6	# b a))))	8 $<$ 9	) b #
7	# b a)))))	8 $>$ 7	b #

関数を使用した場合は，誤りを発見するまで4ステップも余分に実行しなければならなかった．

演習問題　5.4

1.　下記の行列に対する順位関数 f，g を構成せよ．

$$\begin{bmatrix} \gtrdot & & \gtrdot & \gtrdot \\ \gtrdot & & \gtrdot & \\ \lessdot & \doteq & \lessdot & \\ \doteq & & \doteq & \end{bmatrix}$$

2. 定理 5.4.6 を証明せよ．
3. 問題 5.2.6 のプログラムを修正して順位関数も出力するようにせよ．
4. 問題 5.3.2 の構文解析アルゴリズムを変更して順位関係の代りに関数を用いるようにせよ．

5.5 順位文法の構成上の難点

われわれがこの技法をまずここで提示したのはこれが最も簡単な技法であり，以後の章で用いる大部分の構文解析技法を説明しているからである．理論的には，この技法は，健全な，能率のよい技法のように思われるが，実用的には，必ずしも好ましいものではない．事実，このほかのほとんどの技法が，この技法よりも優れている．2個の記号の間には，2個以上の関係が存在することが非常に多い．その際できることは，衝突を避けるために文法にあたってそれを変更することだけである．これによって，文法が読みにくくなるばかりか，言語全体の構造を変えてしまうこともありうる．

左再帰的な場合に問題が生ずる．$U ::= U\cdots$ という規則を仮定しよう．もし別の規則 $V ::= \cdots SU\cdots$ が存在するならば，$S \doteq U$ かつ，$S \lessdot U$ が成立つ．この衝突は，別の非終端記号 W を導入し，中間的な還元の過程を設けることによって除くことができることもある．すなわち

$$V ::= \cdots SU\cdots$$

を

$$V ::= \cdots SW\cdots,\ W ::= U$$

と変更する．W が新しく導入された記号で，これによって $S \doteq W$ かつ $S \lessdot U$ となる．この方法を**層化** (stratification) とか**分離** (separation) と呼んでいる．しかしこの方法はかなり技巧的で，このような衝突が滅多に起らないような，別の構文解析技法を導入しよう．同様な問題は右再帰的な文法における関係 $\gtrdot$ と $\doteq$ でも生ずる．層化はつねにうまくゆくとは限らず，別の衝突をひき起すことが多い．もし $R \lessdot S$ と $R \gtrdot S$ が同時に成立つようなことになると，どうしようもなく，これ以外の技法を用いる方がましである．

層化の説明のために，おなじみの算術式の文法を採り上げよう．

(5.5.1)　E ::= E + T|T
　　　　T ::= T * F|F
　　　　F ::= (E)|i

最初の規則から $+ \doteq$ T であり，T は左再帰的であるから $+ \lessdot$ T である．同様なことは (と E についても成立つ．そこで文の構造をこわさず，言語を変更しないように文法を変更するにはつぎのようにする．

```
E   ::= E1
E1  ::= E1 + T1|T1
T1  ::= T
T   ::= T * F|F
F   ::= ( E )|i
```

つぎに，下記のような関係 FIRST⁺ と LAST⁺ を用いる：

U	**(*U*, *S*) が FIRST⁺ に属するような *S***	**(*U*, *S*) が LAST⁺ に属するような *S***
E	E1, T, T1, F, (, i	E1, T1, T, F,), i
E1	E1, T, T1, F, (, i	T1, T, F,), i
T1	T, F, (, i	T, F,), i
T	T, F, (, i	F,), i
F	(, i	), i

これから順位行列と関数はつぎのようになる

	E	E1	T1	T	F	i	(	+	*	)	*f*	*g*
E										$\doteq$	2	2
E1								$\doteq$		$\gtrdot$	4	3
T1								$\gtrdot$		$\gtrdot$	5	4
T								$\gtrdot$	$\doteq$	$\gtrdot$	6	5
F								$\gtrdot$	$\gtrdot$	$\gtrdot$	7	6
i								$\gtrdot$	$\gtrdot$	$\gtrdot$	7	7
(	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$				2	7
+			$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$				4	4
*					$\doteq$	$\lessdot$	$\lessdot$				6	6
)								$\gtrdot$	$\gtrdot$	$\gtrdot$	7	2

この変更は大したことはなさそうに見えるかもしれないが，もし規則が 100

あり，問題の記号が 100 あるような文法（ALGOL のような言語に相当）では，慣れた人でもこの文法を順位文法に合わせるには長い時間を要する．

読者はここで，次章以下で導入する他の技法が何故これより優れているのかとたずねるかもしれない．順位文法の技法では，把手を決定するのにその前後の文脈のわずかしか利用していないことに注意．実際，決定に際して，隣接する 2 個の記号しか使用していない．もし，これ以外の記号か，あるいは，もっと多くの記号を用いれば，衝突が生ずる機会が減るかもしれない．したがって，把手の後部を検出するために S_i と S_{i+1} を調べる代りに，

$$S_{i-1},\ S_i,\ S_{i+1} \quad や \quad S_i,\ S_{i+1},\ S_{i+2}$$

を調べる．例えば，文法 (5.5.1) の文形式 E＋T＊F を考えてみよう．ここでは，$+ \lessdot \mathrm{T}$，$+ \doteq \mathrm{T}$ という衝突が存在する．したがって，2 個の記号 ＋ と T からは，T が把手の頭部なのかどうか――加算をまず行うべきなのかどうか――を知ることができない．しかし，2 個の記号 ＋ と ＊ から，あるいは，3 個の記号 ＋T＊ から，加算はまだ実行できず，したがって ＋ は把手には含まれないことが直観的にわかる．

演習問題 5.5

1. 下記の文法を，言語を変えずに順位文法に変更し，全ての順位関係を列挙せよ．

   ⟨宣言⟩ ::= ⟨型⟩⟨名前の並び⟩
   ⟨型⟩ ::= REAL | INTEGER | BOOLEAN
   ⟨名前の並び⟩ ::= i | ⟨名前の並び⟩, i

2. 下記の文法を，言語を変えずに順位文法に変更し，全ての順位関係を列挙せよ．

   P ::= ⟨be⟩
   ⟨be⟩ ::= *b* := ⟨be⟩ | ⟨ae⟩ = ⟨ae⟩
   ⟨ae⟩ ::= *a* := ⟨ae⟩ | ⟨at⟩
   ⟨at⟩ ::= ⟨at⟩ − ⟨ap⟩ | ⟨ap⟩
   ⟨ap⟩ ::= *a* | (⟨ae⟩) | @ (⟨be⟩)

(⟨be⟩ は論理式 (Boolean expression)，⟨ae⟩ は算術式 (arithmetic expression)，⟨at⟩ は算術項 (arithmetic term)，⟨ap⟩ は算術 1 次子 (arithme-

tic primary）を示す．"b"（"a"）は，論理（算術）変数名．論理式 $b:=$ 〈be〉を評価するためには，〈be〉を評価して，これを b に代入する．よって，式の値が，b の値となる．算術式 $a:=$ 〈ae〉についても同様である．（〈be〉）の値は，もし〈be〉が真のとき 1，偽のとき 0 である．）

5.6　参考文献概説

順位関係と順位関数の概念を最初に形式化したのは Floyd (63) である．Floyd 流の形式化については次章で学ぶ．本章の形式は Wirth と Weber (66) によった．本章での順位関係の計算法は Martin (68) にある．本章で用いた順位関数の計算法は Bell (69) による．ALGOL W コンパイラでは (Bauer, Becker, Graham (68) 参照)，自動的に構成した順位行列を用いている．単純順位文法を拡張した論文はかなりあるがこれは6章の最後に挙げる文献を参照してほしい．

第 6 章

その他の上向き認識法

前章では，上向き構文解析に対する簡単な順位技法を議論した．本章では，これ以外の上向き解析アルゴリズムをいくつか議論しよう．これらのアルゴリズムは，全て，文が出発記号に還元されるまで，以下のステップを繰返すという意味において同じように動作する．

1. 把手 x（あるいはその変形）を探す．
2. 右辺がxであるような規則 $U ::= x$ を探す．
3. x を U に還元し，構文の木の枝を1本作り出す．

これらの認識法のちがいは2点にある．第1点は，把手を決定するために用いられる記号の個数と位置であり，第2点は，これらの認識アルゴリズム中の規則と表の構造である．それぞれの認識法は，つぎのようにして，計算機に実現される．

1. 表を用いる一般的な認識アルゴリズムをプログラムする．文法は表に記述される．
2. 文法が与えられると，その適法性を検査し，認識アルゴリズムに必要な表を組立てる**自動作成機**（constructor）をプログラムする．
3. 与えられた文法に対する認識アルゴリズムを実現するために，その文法を入力データとして，2.で作った自動作成機を走らせ，その出力結果と1.で作った認識プログラムを合併する．

これで与えられた文法の文を認識するプログラムが得られる．このプログラムは必要に応じて使用することができるが，そのたびに自動作成機を走らせる必要はない．単純順位文法の場合は，図5.4 の流れ図が上の認識プログラムに対応するものであり，順位行列と規則類が表にあたる．

上向き解析法には，初期の頃のコンパイラで用いられた方法を形式化したものがいくつかある．文法に対する要請は，直観的に組立てられた認識のアルゴリズムの働き方によって大部分決定されていた．このことは，演算子順位法でも推移行列法においても同様である．一方 (m, n) 限定文脈法や $LR(k)$ のような方法は，まず理論的に導かれたがまだ実用には供されていない．

以下の各節は，それぞれ特徴のある認識法の説明にあてられている．各節の議論は短かいが完結したものになっている．証明は，演習問題として読者に委ねる．これらの技法は，把手を探し，その把手の前後の文脈を見て，その把手を還元すべき記号を決定するという意味において，全て同じものである．

6.1　演算子順位文法

演算子文法

算術式 $2+3*5$ の値を評価すればだれでも 17 という答にするだろう．何故乗算を先に実行したかと問われれば，乗算は，つねに加算に先立って行う，すなわち，演算子 $*$ は，演算子 $+$ より**高い順位にある**からと答えるだろう．ここで重要なことは，被演算子 2，3，5 は，どの演算を先に行うべきかを決定するに何の役割も演じなかったことである．決定には演算子だけが関与している．単純順位法では**全て**の記号——演算子も被演算子も——間の順位を定義し，これがいろいろの困難をひき起したことを思い出そう．演算子順位法は，演算子間の順位関係だけを形式化したものである．演算子は，実は文法の終端記号なのである．

この方法が，いくつかのコンパイラで普通の算術式の処理のためにどのように用いられてきたかを説明することから始めよう．この構文解析部は，**演算子**用スタック OPTOR と，**被演算子**用スタック OPAND という 2 個のスタックを用意している．OPTOR には，2 項演算子，単項演算子，かっこ，それに各式の前後の区切り記号 # が入る．OPAND には，名前と，そのほか処理が進むにつれて生成される被演算子が入る．これらの 2 個のスタックと並んで，5.4 節の順位関数と同様な，整数値をとる 2 個のベクトル f と g を用意する．

一般的な構文解析アルゴリズムは下記のような段階を踏んで実行される．ここで S_1 は演算子用スタックの最上段にある記号であり，S_2 は，つぎに入って来る記号である．

1.　もし S_2 が名前であれば，これを被演算子用スタックにおき，以下の第

2，第3ステップを省略する．

2. もし $f(S_1) \leq g(S_2)$ ならば，S_2 を演算子用スタックにのせ，つぎの入力記号をとってくる．

3. もし $f(S_1) > g(S_2)$ ならば，S_1 によって決まる（意味付け）ルーチンを呼出す．このルーチンは，意味処理を行い，S_1 をスタックから取除き，場合によっては，さらにほかの演算子も取除く．そして，S_1 に連らなる被演算子を OPAND から取除き，演算子 S_1 の演算の結果を記述している要素を OPAND にのせる．これは，文形式の把手を還元する操作に対応している．

これを算術式の構文解析で説明するために，#A＋B＋C# を採り上げよう．これを行いながら，ベクトル f と g とに適当な値をうめてゆこう．解析では先ず OPTOR に # をのせ，OPAND は空にしておく．第1ステップは，名前 A を OPAND にのせる．第2ステップは当然 ＋ を OPTOR にのせるだろう（＋ を還元したり処理したりするのは，＋ の両側の被演算子が既知になるまでは行うことができない）．これを遂行させるためには，$f(\#) < g(+)$ が必要で，$f(\#) = 1$，$g(+) = 4$ を代入しておこう．

第3ステップは，B を OPAND にのせて，つぎのような状況になる．

OPTOR：| ＋ / # |　　OPAND：| B / A |　記号列の残り：＋C#　　$f(\#)=1$　$g(+)=4$

ここで ＋ が OPTOR の最上段にあり，この被演算子 A と B は OPAND の最上段にあることに注意しよう．ここでは ＋ を意味的に処理して，場合によっては（＋ A, B, T1）のような演算の内部形式を生成したい．そしてつぎに ＋ を OPTOR から取除き，A と B を OPAND から取除き，中間結果 A＋B を意味する T1 を OPAND にのせてやりたい．これによって状況は，

OPTOR：| # |　　OPAND：| T1 |　記号列の残り：＋C#　　$f(\#) = 1$　$f(+) = 5$　$g(+) = 4$

となる．このためには，$f(+) > g(+)$ が必要であり，$f(+) = 5$ と設定する．これは，規則 E ::= E＋T を用いて還元操作を行うことに対応していることに注意しよう．

つぎのステップは，C を OPAND に押込むことで，これで

OPTOR :	OPAND :	記号列の残り：	
+	C		$f(\#)=1$
#	T1	#	$f(+)=5$　$g(+)=4$

となる．再び + が OPTOR にあり，その被演算子は OPAND の最上段にのっているので，再び + の処理——還元——を行いたい．これには $f(+)>g(\#)$ が必要である．このステップを終って最終的にはつぎのようになっている．

OPTOR :	OPAND :	記号列の残り：	
#	T2	#	$f(\#)=1$　$g(\#)=1$
			$f(+)=4$　$g(+)=5$

このようにして，(OPTOR の最上段の要素，入力記号）のそれぞれの対に対して何をさせたいかを見て，それに応じて f と g に値を与えることによって，関数を決める．#A＋B＊C# から，＋ が OPTOR にのっており，＊ がつぎの入力記号であれば，＊ もスタックの上にのせるためには，$f(+)<g(*)$ が必要であることに注意しよう．これには $g(*)=6$ と置いてやればよかろう．関数の値を決めるためのもっと組織的な方法としては，先ず必要な関係を全て数値化し 5.4 節に述べたグラフの方法かブール行列の方法を用いるべきであろう．

誤りの検出やマイナス記号が2項演算子なのか単項演算子なのかを決めるために被演算子が各演算子やかっこに続いているかいなかを注意深く検査しなければならない．このためには，OPTOR の各要素に1ビットを設ける（初期値はゼロにしておく）．被演算子が OPAND にのせられるときはいつでも，OPTOR の最上段にある記号のこの1ビットを1にセットして，このあとに被演算子が続いていることを示す．

この方法は，**2 重優先** (double priority) ということばでも知られている．何となれば，いずれの演算子の優先順位が高いかを決定するのに二つの関数が用いられているからである．唯一つの関数 h だけでもよいと考える人もあるかも知れない．もし $h(S_1)>h(S_2)$ ならば，スタック上で還元を行い，反対の場合は S_2 をスタックにのせる．しかし，式にかっこを用いていると，この方法は不可能になる．#(A)＊B# から $h(\,)\,)>h(*)$ が必要となり，#(A＊B)# から，$h(*)>h(\,)\,)$ が必要となるが，これは不可能である．

OPTOR 上で“還元”が行われるとき，最上段の記号が，何らかの方法で実行すべきサブルーチンを決定している．OPTOR 中の記号 S の一つの表現として

$f(S)$	実行すべきサブルーチンの番地

のようなものがある．今もってコンパイラをアセンブリ言語でプログラムする理由の一つに，普通の高水準のプログラム用言語ではこのような能率的な表現法を採用できないことがある．

上のように処理過程の概要をいきなり示したのは，普通のコンパイラ作成者がどのように行っているかを示すのが目的だからである．ここでもう少し詳しくこの技法の形式化を議論しよう．この議論では，スタックは2個ではなく1個だけを用いることにする．もっとも，読者は，後になれば，スタックを1個にするか2個にするかは，まさに趣味の問題であることがわかるだろう．

ここで，文法を少し制限して，文形式中の演算子の間に無制限な個数の被演算子（非終端記号）が現れることがないようにする．云い換えれば，式は挿入記法でなければならない．

(6.1.1) **定義**　文法 G のどの規則も $U ::= \cdots VW \cdots$ という形式でないならば，この文法は**演算子文法** (operator grammar; 略して **OG**) と呼ばれる．ここで V, W は非終端記号である．■

したがって，文法 OG の規則の右辺には非終端記号が隣合うことはない．

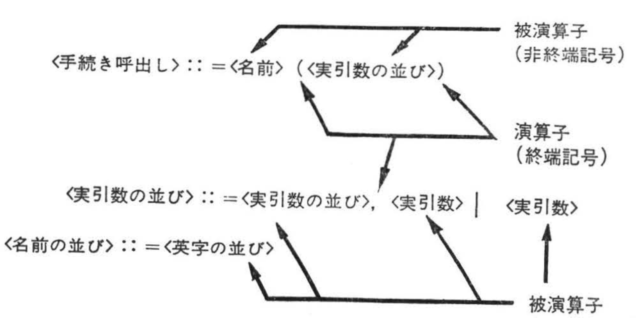


図6.1　演算子と被演算子

これは，文形式についてもいえる．すなわち，OG の文形式は隣合った2個の非終端記号を含まないことが証明できる（演習問題1）．われわれが取扱う文法はさほど困難なく OG に変換することができるものが大部分である．

この章では，**演算子**と**終端記号**を同じ意味で，適宜使用する．**被演算子**と**非終端記号**についても同様である．図6.1 はいくつかの規則における被演算子と演算子を示している．

演算子順位文法

下記の関係は単純順位関係と全く同様である．唯一の違いは，再び演算子間のみの関係を定義している点である．

(6.1.2)　**定義**　　文法 G は OG とし，R と S は任意の演算子（終端記号）としよう．すると適当な非終端記号 V，W に対して

1. $R \doteq S$ が成立つのは，規則 $U ::= \cdots RS \cdots$ あるいは $U ::= \cdots RVS \cdots$ が存在し，そしてそのときに限られる．
2. $R \lessdot S$ が成立つのは，規則 $U ::= \cdots RW \cdots$ ($W \stackrel{+}{\Rightarrow} S \cdots$ あるいは $W \stackrel{+}{\Rightarrow} VS \cdots$) が存在し，そしてそのときに限られる．
3. $R \gtrdot S$ が成立つのは，規則 $U ::= \cdots WS \cdots$ ($W \stackrel{+}{\Rightarrow} \cdots R$ あるいは $W \stackrel{+}{\Rightarrow} \cdots RV$) が存在し，そしてそのときに限られる．■

(6.1.3)　**定義**　　演算子文法 OG の任意の2個の終端記号の間に $\lessdot$，$\gtrdot$ および $\doteq$ の高々1個の関係が成立てば，この文法を**演算子順位文法** (operator precedence grammar; 略して **OPG**) と呼ぶ．■

(6.1.4)　例　　文法 G[E] を考えよう：

$$
\begin{aligned}
\mathrm{E} &::= \mathrm{E} \;+\; \mathrm{T} \mid \mathrm{T} \\
\mathrm{T} &::= \mathrm{T} \;*\; \mathrm{F} \mid \mathrm{F} \\
\mathrm{F} &::= (\;\mathrm{E}\;) \mid \mathrm{i}
\end{aligned}
$$

3行目の規則から($\doteq$)が得られる．これが関係 $\doteq$ としては唯一のものである．何となれば，ほかのどの右辺にも終端記号を2個含むものがないからである．$\mathrm{E} \Rightarrow \mathrm{E}+\mathrm{T} \Rightarrow \mathrm{T}+\mathrm{T} \Rightarrow \mathrm{T}*\mathrm{F}+\mathrm{T} \Rightarrow \mathrm{F}*\mathrm{F}+\mathrm{T} \Rightarrow \mathrm{i}*\mathrm{F}+\mathrm{T}$ だから，規則 F ::= (E) から（$\lessdot$+，（$\lessdot$*，（$\lessdot$i が得られる．同じ規則と直接導出の系列から ＋$\gtrdot$) も示すことができる．完全な演算子順位行列は図6.2に示す．2個の終端記号の間には高々1個の関係が成立つのであるから，これは

	+	*	(	)	i
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
)	$\gtrdot$	$\gtrdot$		$\gtrdot$	
i	$\gtrdot$	$\gtrdot$		$\gtrdot$	

図6.2 文法 (6.1.4) に対する順位行列

OPG である.■

関係の構成

関係 $\doteq$ を構成するのに詳しい説明は不要だろう. 規則の右辺に RVS か RS の型があるのを探すだけでよい. 関係 $\lessdot$ を構成するには, 各 W に対して $W \stackrel{+}{\Rightarrow} S\cdots$ あるいは $W \stackrel{+}{\Rightarrow} VS\cdots$ (V は非終端記号) となるような終端記号 S の集合を探す何らかの方法を見付けなければならない (関係 $\gtrdot$ の構成についても同様). このために二つの関係を新しく定義しよう.

(6.1.5) **定義** G を OG とし, S を終端記号とする. このとき U FIRSTTERM S が成立つのは, 規則 $U ::= S\cdots$ あるいは $U ::= VS\cdots$ が存在し, しかもそのときに限られる.

U LASTTERM S が成つのは, 規則 $U ::= \cdots S$ あるいは $U ::= \cdots SV$ が存在し, しかもそのときに限られる.

ただし V は任意の非終端記号とする.■

"FIRSTTERM" ("LASTTERM") は, 規則の右辺の最初の (最後の) 終端記号を表している. これらの関係を規則から構成するのは明らかに容易である. 5種類の関係 $\doteq$ (単純順位関係), FIRST, FIRSTTERM, LAST, LASTTERM (これらの関係を, ブール行列として構成する方法は全てわかっている) を用いて, 関係 $\lessdot$ と $\gtrdot$ を構成する.

(6.1.6) **定理** $\lessdot$ は三つの関係 $\doteq$, FIRST*, FIRSTTERM の積である: $\lessdot$ = ($\doteq$) (FIRST*) (FIRSTTERM). 同様に $\gtrdot$ = TRANSPOSE ((LAST*) (LASTTERM)) ($\doteq$).■

(証明は簡単であり読者に演習問題3としておく.) これによって関係 $\lessdot$ (お

よび ⋗) を簡単に計算する方法が得られた．規則から三つの関係に関するブール行列を構成し，その一つについて反射的推移的閉包を作って，三つの行列を掛け合せればよい．

素句——その検出と還元

ただ1個の非終端記号からなる把手は，これらの関係を用いては見出すことができない．それは，これらの関係からは，非終端記号が，見えないからである．例えば，例 6.1.4 の文法における文形式 F＋T を考えてみよう．把手は F であるが，われわれの関係は，# ⋖ ＋ と ＋ ⋗ # のみである（ここで再び文形式は # で前後を区切られ，全ての終端記号 S に対して # ⋖ S, S ⋗ # と仮定する）．演算子順位技法では，したがって，すでに定義したような正準な構文解析を構成することは不可能であろうが，それにかなり似通ったものは計算できるであろう．構文解析は，依然として上向き法によるのであるが，厳密に左から右というわけにはゆかない．解析の各段で，最も左側の**素句** (prime phrase) と呼ぶものを認識しそれを還元するのである．

(6.1.7)　**定義**　　G は出発記号を Z とする文法とする．文形式の**素句**とは，少なくとも1個の終端記号を含み，自分自身以外の素句を含まない句をいう．■

文法 (6.1.4) において # で前後を区切られた文形式 #T＋T∗F＋i# を考えよう．この文形式の句は，T＋T∗F＋i, T＋T∗F, i, 左側の T, T∗F である（これは，図6.3 の文形式の唯一つの構文の木からわかる）．このうちどれが素句であろうか．素句は，終端記号を含む別の句を含むことができないので，i を含んだ T＋T∗F＋i は除外する．同様に T＋T∗F も句 T∗F を含む．可能性は T, T∗F それに i に残された．T は，終端記号を含まないから素句にはなり得ない（T が文形式の把手ではあっても素句でないことに注意）．したがって，素句は T∗F と i である．

しばらくのあいだ，一般的な文形式を # で区切ってつぎのように書く：

$$\# N_1T_1N_2T_2\cdots N_nT_nN_{n+1}\#$$

ここで N_i は非終端記号であり存在することもあるし存在しないこともある．T_i は終端記号である．換言すれば，文形式は，n 個の終端記号と隣り合う2個の終端記号の間に高々1個の非終端記号をもつ．OG の任意の文形式がこの

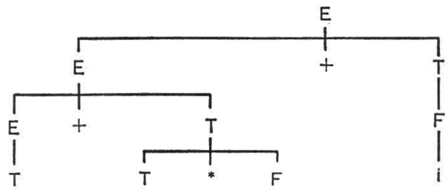


図6.3　T＋T∗F＋i に対する構文の木

形式であったことを想起しよう．演習問題 6～8 は，以下の定理の証明にあててある．

(6.1.8)　**定理**　　文法 OPG の文形式の最も左側の素句とは

(6.1.9)　$T_{j-1} \lessdot T_j,\ T_j \doteq T_{j+1},\ \cdots,\ T_{i-1} \doteq T_i,\ T_i \gtrdot T_{i+1}$

となるような最も左側の部分記号列 $N_jT_j \cdots N_iT_iN_{i+1}$ である．■

この定理は，単純順位文法の対応する定理とそのまま同じようにみえるが，ここでの主な違いは，関係から非終端記号が除外されていることである．T_j のすぐ左側，あるいは T_i のすぐ右側，にある非終端記号は，つねに素句に含まれることに注意しよう．

この定理の説明のために，図 6.2 の順位行列を用いて，文形式 T＋T∗F＋i の構文解析をお目に掛けよう．読者は，図 6.3 の構文の木に従って，この構文解析をつぶさに調べてほしい．下の表は，それぞれの素句が還元されるべき記号を示している．もっとも，この記号の決め方はまだ説明していない．これは，すぐつづけて議論する．

ステップ	文 形 式	関　係	素 句	還元すべき記号
1	#T＋T∗F＋i#	$\# \lessdot + \lessdot * \gtrdot +$	T∗F	T
2	#T＋T＋i#	$\# \lessdot + \gtrdot +$	T＋T	E
3	#E＋i#	$\# \lessdot + \lessdot i \gtrdot \#$	i	F
4	#E＋F#	$\# \lessdot + \gtrdot \#$	E＋F	E

上の構文解析には，それぞれの素句を還元すべき非終端記号が示してある．しかしながら，非終端記号は，還元されるべき最も左側の素句を探し出すのに全然用いられなかったことに注意しよう．したがって，スタック上に，正しい非終端記号がのっていようといまいと，違いが全くないのである．実際，非終

端記号は完全に除いてしまうこともできる.

コンパイラでは，それぞれの非終端記号あるいは被演算子に多くの意味情報——型とか，番地とか，仮引数であるとか——が結合されているのが普通である．第1章に簡単にふれたように，素句を処理する意味付けルーチンが必要となろう．意味付けルーチンの関心は，記号に結びついた意味情報に対してだけであって，記号そのものには何の関心もない．例えば，T＊F を還元する意味付けルーチンは，それが T＊F であれ，F＊T であれ，F＊F であれどうでもよい．このルーチンは，各非終端記号に含まれる意味——型，番地など——だけしか使用しない．2.4 節から，非終端記号 T とは，この文法をあいまいでなくし演算子に順位を与えるためにのみ用いられていたことを想起すべきである．意味のために用いたのではなかった.

もし，形式的な構文解析アルゴリズムが必要であれば，素句を還元すべき非終端記号を明らかにしてやらなければならないことに注意．しかし，われわれに必要なのは，形式ではなくて，実用性である．各文形式に対する構文の木が一意的であるとは，云ってこなかったことに注意．構文の木は一意的でないかもしれない．しかし，非終端記号にあたる木の節点にある名前を別にすれば，木の枠組 (structure) 自体は一意的であると云うことができる．これは，構文解析の各段において，最も左側の句が一意に定まることから明らかである.

もちろん，意味付けルーチンには，もっと重みがかけられている．これらのルーチンでは，単純順位文法のとき以上に被演算子の検査を行わなければならない．それでも，こちらの認識法の方がずっと能率がよいので，なお，いくらかの利がある．記憶容量についても，順位行列が小さくなるので節約できることに注目しよう．この行列は，終端記号の個数だけの行と列でよい．さらに，この認識法では，E ⇒ T とか T ⇒ F に対応するような還元を陽には行わない．つまり，T や F が素句でないからである．このような還元操作に対しては，何の意味付けも発動されないので，かえって好都合である.

図6.4 に OPG 認識アルゴリズムの流れ図を示した．ここでいつものように，スタック S とその指標 i を用い，入力記号列は $T_1 \cdots T_n$ であり，その指標は k，そのほか，整数変数 j と，記号を値とする2個の変数 P, Q を用いている.

演算子順位関係から順位関数を作り，5.4 節と同じようにこの関数を使用することができる．そして本節の最初に特別な方法で説明したことを形式的に実行している.

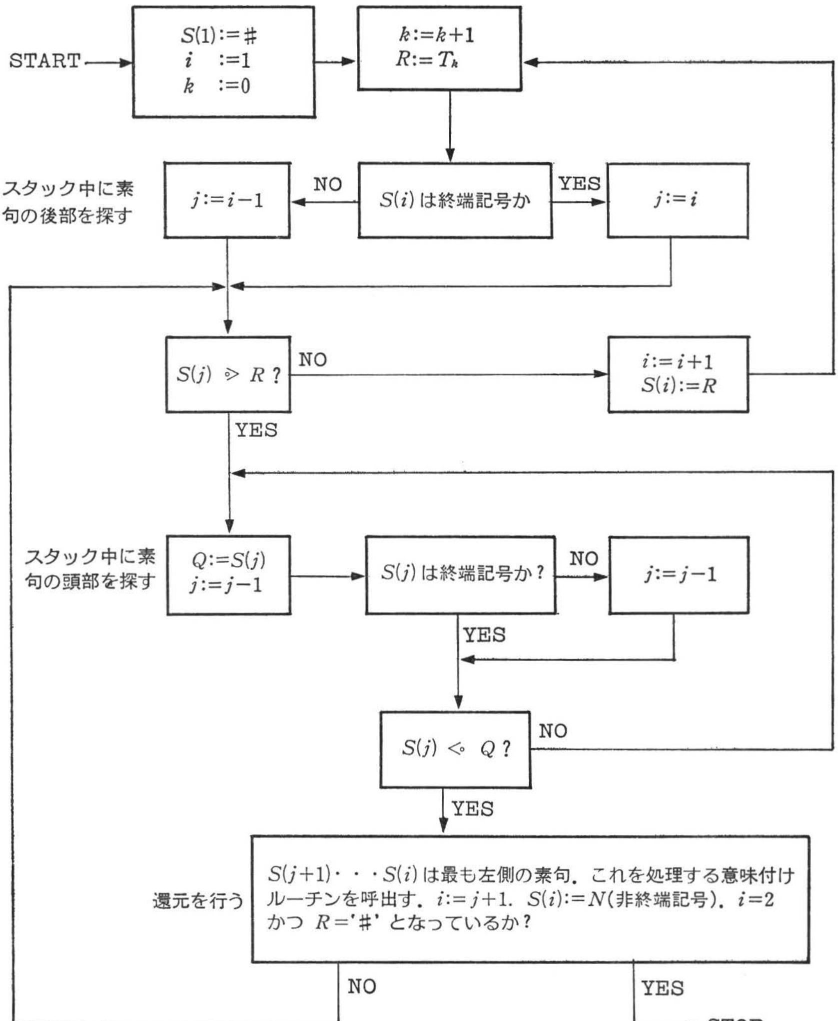


図 6.4　演算子順位技法による認識法

例として，文法 (6.1.4) の文 i∗(i＋i) の構文解析は，つぎのようになる．

ステップ	S(1)…	関係	R	T(k)…
0	#	⋖	i	∗(i＋i)#
1	#i	⋗	∗	(i＋i)#
2	#N	⋖	∗	(i＋i)#
3	#N∗	⋖	(	i＋i)#
4	#N∗(	⋖	i	＋i)#
5	#N∗(i	⋗	＋	i)#
6	#N∗(N	⋖	＋	i)#
7	#N∗(N＋	⋖	i	)#
8	#N∗(N＋i	⋗	)	#
9	#N∗(N＋N	⋗	)	#
10	#N∗(N	≗	)	#
11	#N∗(N)	⋗	#	
12	#N∗N	⋗	#	
13	#N	STOP		

演習問題　6.1

1. OG では，文形式中で，2個の非終端記号が隣合うことがないことを証明せよ．**ヒント**：文形式の導出において，直接導出の数に関する帰納法で証明せよ．

2. G をある文法とする．もし，規則の右辺に2個の非終端記号が隣合うとすれば，少くとも1個の文形式において，2個の非終端記号が隣合うことを証明せよ．**ヒント**：全ての文法は既約になる（定義 2.8.4 参照）．

3. 定理（6.1.6）を証明せよ．

4. 例（6.1.4）の文法に対して，諸関係，≗，≐，FIRST, FIRSTTERM, LAST および LASTTERM に対するブール行列を作り上げよ．つぎに，FIRST および LAST からそれぞれ FIRST*, LAST* を計算せよ．最後に，定理 6.1.6 を用いて，関係 ⋖ および ⋗ を計算せよ．

5. 例（6.1.4）の文法の下記の文形式の句と素句を列挙せよ．

   E, T, i, T∗F, F∗F, i∗F, F∗i, F＋F＋F.

6. もし T がある文形式の句に含まれていれば，T の直前（直後）にある非終記端号も，その句に含まれることを証明せよ．**ヒント**：もし含まれないならば，2個の非終端記号が隣合う文形式を作れ．

7. 任意の素句は (6.1.9) を満たすことを証明せよ（文形式の前後は # で区切られ，任意の終端記号 S に対して $\# \lessdot S$, $S \gtrdot \#$ と仮定する).

8. (6.1.9) を満たす文形式の任意の部分記号列 $N_iT_i \cdots N_jT_jN_{j+1}$ は素句であることを証明せよ．**ヒント**：定理 (5.2.8) の証明を例として用いよ．

9. 文法 (6.1.4) に対する 図 6.2 の順位行列と 図 6.4 の認識アルゴリズムを用いて，次の各文を構文解析せよ： i, i + i, i ∗ i + i, i ∗ (i ∗ i), i ∗ (i + i ∗ i) + ((i + i) ∗ i).

6.2 高次の順位技法

(1,2) (2,1) 順位技法

この技法は，単純順位技法を 2 段階拡張してもっと実用的な認識法に近づけるものである．これは，さらに多くの文法で使用できるものである．第一に，把手の頭部を見つける問題と後部を見つける問題を分離する．後部を探すときには，関係 $\gtrdot$ を探す．$R \lessdot S$ と $R \doteq S$ の関係が成立しても，ここでは，R が後部になるかどうかが知りたいのであるから，実際には問題にならない．したがって，ここでは $\gtrdot$ と $\leqq$ の関係だけを使用する．同様に，頭部を探すときは，$\lessdot$ と $\geqq$ の関係だけを使用する．

つぎの段階は，把手の頭部と後部を検出するのに，2 個ではなく，3 個の記号を使用する．つぎのような諸関係を定義する．

(6.2.1) **定義**　G を文法，R, S, T を記号としてつぎのように定義する．

$RS \circledgtr T$ ： 正準な文形式 $\cdots RST \cdots$ が存在して，S が把手の後部の場合．

$RS \circleddash\!\!\leqq T$ ： 正準な文形式 $\cdots RST \cdots$ が存在して，T が把手に含まれる場合．

$R \circledless ST$ ： 正準な文形式が存在して S が把手の頭部である場合．

$R \circleddash\!\!\geqq ST$ ： 正準な文形式が存在して R が把手に含まれる場合．■

(1,2) (2,1) 順位とは，把手の頭部と後部を検出するために用いられる記号の数を示している．後部を検出するために，3 個の記号 R, S, T について，$RS \circledgtr T$ または $RS \circleddash\!\!\leqq T$ の関係を調べる．これが (2,1) の由来である．一

たん後部となる記号が見つかると頭部を見つけるためにそこから左側へ戻って $R \textcircled{<} ST$ あるいは $R \textcircled{\ge} ST$ という関係を調べにゆく．そこで，(1,2) という．

単純順位技法の場合に比べて衝突の回数は少なくなる．算術式に対する文法

(6.2.2)　$\mathrm{E} ::= \mathrm{T} | \mathrm{E}+\mathrm{T} \qquad \mathrm{T} ::= \mathrm{F} | \mathrm{T}*\mathrm{F} \qquad \mathrm{F} ::= \mathrm{i} | (\mathrm{E})$

で $+ \doteq \mathrm{T}$ と $+ \lessdot \mathrm{T}$ とが得られる．これは，T に続く記号を用いれば，$+ \textcircled{\ge} \mathrm{T}+$ と $+ \textcircled{<} \mathrm{T}*$ に分解することができる．

(6.2.1)に与えた諸関係は，導出を用いてつぎのように定義し直すことができる（Z は出発記号．大かっこで括られた部分記号列は存在しないこともある）．

$RS \textcircled{>} T$ ：$Z \overset{*}{\underset{c}{\Rightarrow}} \cdots[R]UTx \underset{c}{\Rightarrow} \cdots RSTx$ という導出が存在する場合．

$RS \textcircled{\le} T$ ：$Z \overset{*}{\underset{c}{\Rightarrow}} \cdots[R][S]Ux \underset{c}{\Rightarrow} \cdots RST\cdots x$ という導出が存在する場合．

$R \textcircled{<} ST$：$Z \overset{*}{\underset{c}{\Rightarrow}} xRU[T]\cdots \underset{c}{\Rightarrow} xRST\cdots$ という導出が存在する場合．

$R \textcircled{\ge} ST$：$Z \overset{*}{\underset{c}{\Rightarrow}} xU[S][T]\cdots \underset{c}{\Rightarrow} xRST\cdots$ という導出が存在する場合．

与えられた文法から諸関係を計算するアルゴリズムを導びく諸定義のために演習問題 1～5 を参照．ここでは，文法 (6.2.2) の順位関係の**いくつか**を示そう：

+T	$\textcircled{>}$	)	((	$\textcircled{\le}$	T	(	$\textcircled{<}$	E+	(	$\textcircled{\ge}$	E)
(F	$\textcircled{>}$	)	((	$\textcircled{\le}$	F	(	$\textcircled{<}$	F+	T	$\textcircled{\ge}$	+T
+F	$\textcircled{>}$	)	((	$\textcircled{\le}$	(	(	$\textcircled{<}$	i+	F	$\textcircled{\ge}$	+F
*F	$\textcircled{>}$	)	((	$\textcircled{\le}$	i	(	$\textcircled{<}$	T*	i	$\textcircled{\ge}$	+F
(i	$\textcircled{>}$	)	(E	$\textcircled{\le}$	)	(	$\textcircled{<}$	F*	T	$\textcircled{\ge}$	*F
+i	$\textcircled{>}$	)	(E	$\textcircled{\le}$	+	(	$\textcircled{<}$	i*	F	$\textcircled{\ge}$	*F
E)	$\textcircled{>}$	)	(E	$\textcircled{\le}$	)	+	$\textcircled{<}$	(E	i	$\textcircled{\ge}$	*F
+T	$\textcircled{>}$	+	+T	$\textcircled{\le}$	*	+	$\textcircled{<}$	T*	+	$\textcircled{\ge}$	T+

ここで (1,2) (2,1) 順位文法を定義することができる．

(6.2.3)　**定義**　(1,2) (2,1) 順位文法 G とは，つぎの条件を満たす文法をいう．

1. 規則の右辺は全て一意的である.
2. 任意の記号 R, S, T に対して, $RS \textcircled{$>$} T$ と $RS \textcircled{$\leqq$} T$ がともに成立つことはない.
3. 任意の記号 R, S, T に対して, $R \textcircled{$<$} ST$ と $R \textcircled{$\geqq$} ST$ がともに成立つことはない.■

(1,2) (2,1) 順位文法があいまいではなく, 任意の正準な文形式 $S_1S_2\cdots S_n$ の把手はつぎの条件

$$S_{j-1} \textcircled{$<$} S_jS_{j+1}$$
$$S_j \textcircled{$\geqq$} S_{j+1}S_{j+2},\ \cdots,\ S_{i-1} \textcircled{$\geqq$} S_iS_{i+1}$$
$$S_{i-1}S_i \textcircled{$>$} S_{i+1}$$

を満たす最も左側の系列 $S_j\cdots S_i$ であることを証明することができる (演習問題 6). 単純順位文法の場合と同様に, 構文解析すべき記号列の最初と最後については要注意である. ここでの関係は3個の記号についてなので, 各記号列の前後に 2 文字記号を置く必要があろう. さらに, 文法の任意の記号 S_1, S_2 に対して

$$(6.2.4)\quad S_1S_2 \textcircled{$>$} \#\qquad \#S_1 \textcircled{$>$} \#\qquad \# \textcircled{$<$} S_1S_2\qquad \# \textcircled{$<$} S_1\#$$

がつねに成立つとする必要があろう.

(1,2) (2,1) 順位文法に対する認識法は, 単純順位文法に対するものと非常に似通っており, 図 5.4 と同じような流れ図になる. 箱 1 は, 上に書いたような区切り記号を考慮していくぶん変更しなければならない. 箱 3 では, 関係 $\textcircled{$>$}$ と $\textcircled{$\leqq$}$ をおさめた3次元配列 PT を用いて $S(i-1)S(i) \textcircled{$>$} R$ を検査することになろう. 一方箱 6 は, 関係 $\textcircled{$<$}$ と $\textcircled{$\geqq$}$ をおさめた3次元配列 PH を用いて $S(j-1) \textcircled{$<$} S(j)S(j+1)$ を検査するように変更される (箱 3 では $S(i+1)$ は R に代入されていることに注意).

われわれが用いる形式言語の文法は大部分 (1,2) (2,1) 順位文法に近いもので, その文法を第一線に引き出すには二, 三の変更しか要しない. しかしながら, この技法は, 実用的には使いものにならない. 何となれば, この技法では二つの3次元配列を用いるからである. 100 個の記号からなる文法では, 2 個の $100 \times 100 \times 100$ 配列が必要である! これらの配列は, 粗(ほとんど 0 要素ばかり) であるのが普通であるから, 関係が成立する R, S, T の組だけを, リストにして保持して, 記憶場所を節約することもできるかもしれない

が，このリストすら禁止的な大きさになってしまう可能性がある．以下，本節は，文法とアルゴリズムを実用的なものにするための取扱い法を示すことにあてよう．McKeeman ほか (70) によって展開された技法でその全容は無理であるが，起りうる可能性を示すにはこれで十分である．

(1,2) (2,1) 順位技法の実用化

ステップ 1　関係 ⋖ および ≧ をもつ配列 PH の除去． 配列 PH は，一たん把手の後部が見つかると，頭部を発見するために用いられる．大部分の把手にとってこの配列は必要ない．何故ならば，スタックの上方にある部分記号列は，規則の右辺と一致してしまうからである．問題が生ずるのは，ある右辺が，他の規則の右辺の，真の後部になっている場合だけである．もし $U ::= ux$ で $V ::= x$ という規則があったときに，スタックの上方に ux が入っている場合，いずれの還元を行えばよいのだろう．これに対して，長さ 2 以上の右辺をもつ規則に対しては，つぎの定理で解決する（証明は読者に任せる）．

(6.2.5)　**定理**　G を (1,2) (2,1) 順位文法，S_i を少くとも 2 個の記号を含む把手の最後の記号とする．すると完全な把手は，規則の右辺である部分記号列 $S_j \cdots S_i$ の**最長の**ものである．■

かくて，$U ::= uX$ と $V ::= X$ という 2 個の規則の場合だけを考えればよい．このような場合の $R ⋖ XT$ となる (R, X, T) の組のリストは全て保存することができる．構文解析中に，$RX ⋗ T$ が生じると，(R, X, T) のリストを検査する．もし，見つかれば，規則 $V ::= X$ を使用する．なければ，スタックの上方と一致するような最長の右辺を持った規則を使用する．

このようにして配列 PH を使用しなくて済んだ．その代りに，全ての右辺に対して最長一致を調べなければならなくなった．もちろん，この問題も議論しよう．

ステップ 2　配列 PT の不要部分の除去． 配列 PT は把手の後部を検出するために用いられる．文の正準な構文解析を実行しているのであるから，任意の把手のすぐ右側にある記号は，終端記号である．したがって，非終端記号に対応するような配列 PT の全ての列は除去することができる．これで，配列はほぼ半分の大きさになる．

ステップ 3　可能なかぎり単純順位関係を用いる． (2,1) 関係 ⋗⃝ と ≦⃝ は，あまりにも記憶場所が必要なのに対して，(1,1) 関係 $\gtrdot$ と $\leqq$ も，われわれのプログラム用言語に対して十分ではないし，融通性に欠ける．そこで，可能な限り単純順位関係を用い，衝突が生じたとき (2,1) 関係に切換えるように工夫する．これには，2次元の配列 P を用いる．P の値は0（関係がない)，1 ($\leqq$)，2 ($\gtrdot$)，3 ($\leqq$ および $\gtrdot$ がともに成立）である．$P[i, j]=3$ に対して 4-組 (S_k, S_i, S_j, q) のリスト TQ を持つ．ここで q は S_kS_i ≦⃝ S_j のとき $q=1$, S_kS_i ⋗⃝ S_j のとき $q=2$ である．

(1,2) (2,1) 構文解析アルゴリズムの改訂

図6.6 は，認識アルゴリズムの改訂版である．使用している変数，表は，下記のとおり．表 P, TQ, TH および $PROD$ は自動作成機が組立てる．

1. S はスタックで i はその指標.
2. $T_1 \cdots T_n$ は入力記号列で，k が記号に対する指標.
3. P は $n \times m$ 行列で n は記号の数，m は終端記号の数.
   $P[i, j]=0$: S_i と S_j の間に関係がない.
   $P[i, j]=1$: $S_i \leqq S_j$
   $P[i, j]=2$: $S_i \gtrdot S_j$

P	#	+	*	(	)	i
E	$\gtrdot$	$\leqq$			$\leqq$	
T	$\gtrdot$	$\gtrdot$	$\leqq$		$\gtrdot$	
F	$\gtrdot$	$\gtrdot$	$\gtrdot$			
#				$\lessdot$		$\lessdot$
+				$\leqq$		$\leqq$
*				$\leqq$		$\leqq$
(				$\leqq$		$\leqq$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
i	$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	

TQ　空　($\gtrdot$ と $\leqq$ の衝突はない)

TH　(#, T, #)　(#, T, +)　((, T, +)　((, T,))　(#, F, #)　(+, F, #)
　　(#, F, +)　((, F, +)　(+, F, +)　((, F,))　(+, F,))　(#, F, *)　((, F, *)　(+, F, *)

スタックの最上段にある記号	T	F	)	i
該当する規則	E ::= T E ::= E+T	T ::= F T ::= T*F	F ::= (E)	F ::= i

図6.5　文法 (6.2.2) に対する (1,2) (2,1) 表

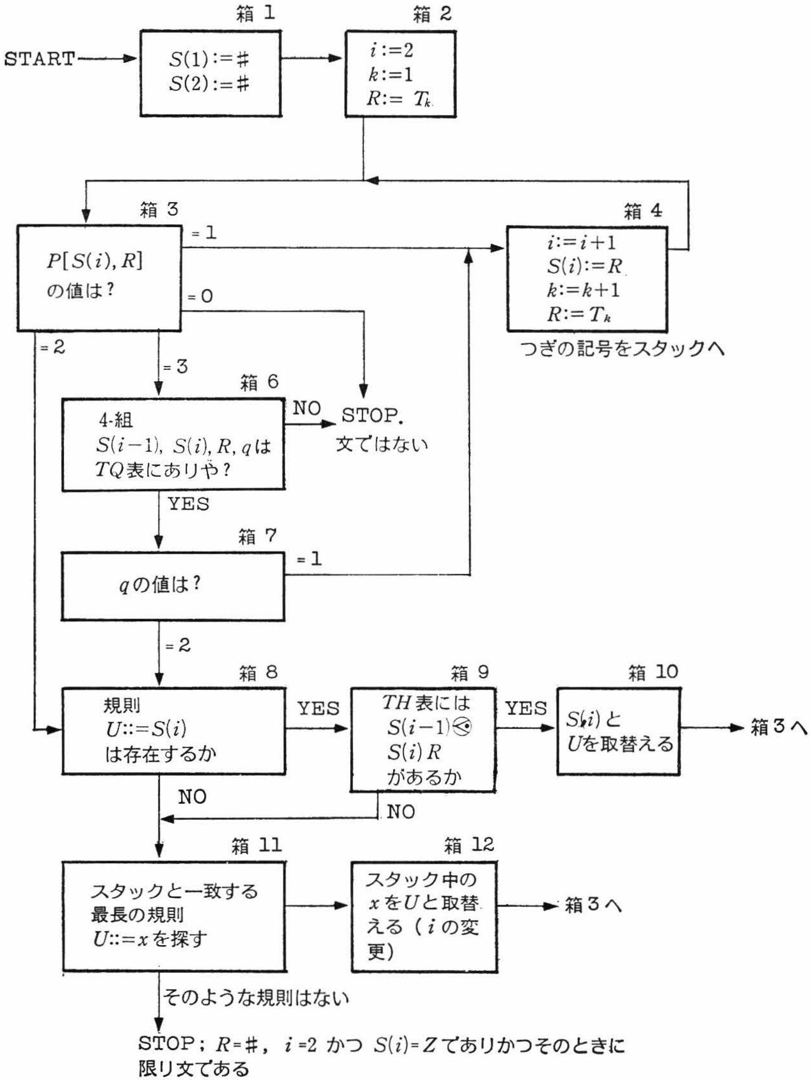


図6.6　(1,2) (2,1) 順位技法による認識アルゴリズム

$P[i, j]=3 : S_i \lessdot S_j$ かつ $S_i \gtrdot S_j$

4. TQ は，4-組 (R, S, T, q) の一覧表である．R と S は記号で，T は終端記号，$q=1$ または 2 である．

   (R, S, T, 1) は $S \lessdot T$, $S \gtrdot T$, RS ⩿ T の場合．

   (R, S, T, 2) は $S \lessdot T$, $S \gtrdot T$, RS ⪀ T の場合．

5. $RROD$ は，規則の一覧表．取り出しを速くするために，右辺の最後の記号で分類しておくべきである．

6. TH は 3-組の一覧表で，右辺が 1 記号からなる規則を使用するかどうかを決定するために用いられる．したがって，3-組 (R, X, T) が一覧表の中にあるのはつぎの場合である：T が終端記号の場合，規則 U ::= X のほかに V ::= $\cdots X$ という規則が少くとも一つある場合，R ⪡ XT が成立つ場合．

例として，文法 (6.2.2) に必要なテーブルを 図6.5 に示した．

(m, n) 順位

ある特定の文法に関して，(1,1) 順位技法でだめ，(1,2) (2,1) 技法でも不十分となれば，一意的な関係を求めてさらに多くの文脈を見ることもできると考えるかもしれない．これが正しい文法も多くある．ここではこの方法を概観したい．しかしながら，ここで概説する機構は，必要となる順位行列の大きさを考えると実用にはならないことを銘記すべきである．(1,2) (2,1) 順位法の場合にしたようにあらゆる工夫を重ねてみても，実際には，役に立たない．

ある文法に対する (m, n) 順位関係を構文の木によってつぎのように定義しよう：$m>0$, $n>0$ を整数とする．x, y を任意の記号列とし $|x|=m$, $|y|=n$ とする．するとつぎの関係が成立する．

$x \lessdot y$: 正準な文形式 $\cdots xy \cdots$ が存在して，y の先頭の記号が把手の先頭の記号である場合．

$x \doteq y$: 正準な文形式 $\cdots xy \cdots$ が存在して，y の先頭の記号と x の最後の記号がともに把手に含まれる場合．

$x \gtrdot y$: 正準な文形式 $\cdots xy \cdots$ が存在して，x の最後の記号が把手の最後の記号である場合．

(m, n) 順位文法とは，つぎの条件を満たす文法をいう：

1. 規則の右辺は全て一意的である．
2. 任意の記号列 x, y($|x|=m$, $|y|=n$ とする）の対に対して，三つの (m, n) 順位関係の高々一つが成立つ．

単純順位文法は，(1, 1) 順位文法と同じものである．ここで (m, n) 順位文法は，あいまいでないこと，正準な文形式の把手は，この順位関係によって一意的に決定されることなども示すことができよう．これは読者に任せよう．困難な部分は，もちろん，構成が可能なように関係を定義することである．

関係を構成する方法の一つは，十分な数の構文の木を調べることである．すなわち，各構文の木から関係を探し，全ての関係が見つかったと確信できるまで続ける．これは，実際 McKeeman ほか (70) が (1, 2) (2, 1) 順位文法の場合に用いた方法である．

演習問題　6.2

以下の問題にはつぎの諸定義を用いるものがある．

もし，規則 U_1 ::= $\cdots SV \cdots$ と関係 V FIRST* U が存在するならば，S は U の**許容前置子** (allowed predecessor) と称し，U AP S と書く．

もし V LAST* U かつ W FIRST* S となるような規則 U_1 ::= $\cdots VW \cdots$ が存在すれば S は U の**許容後置子** (allowed successor) と称し，U AS S と書く．

1. 関係 AP は AP := (≐)(FIRST*) で定義されることを証明せよ．

2. $RS \textcircled{>} T$ が成立つのは以下の条件の一つが成立ちしかもそのときに限られることを示せ：

   a) 規則 U ::= $VW \cdots$ が存在し，U AP R, $V \overset{+}{\Rightarrow} S$ かつ W FIRST* T である．

   b) 規則 U ::= $\cdots VW \cdots$ が存在し，$V \underset{c}{\overset{+}{\Rightarrow}} \cdots RS$ かつ W FIRST* T である．

   c) 規則 U ::= $\cdots RVW \cdots$ が存在し，$V \overset{+}{\Rightarrow} S$ かつ W FIRST* T である．

3. $RS \textcircled{$\leqq$} T$ が成立つのは以下の条件の一つが成立ちしかもそのときに限られることを示せ：

a) 規則 $U ::= SW\cdots$ が存在し，U AP R かつ W FIRST* T である．

b) 規則 $U ::= \cdots RSW$ が存在し，W FIRST* Tである．

4. $R \lessdot ST$ が成立つのは以下の条件の一つが成立し，しかもそのときに限られることを示せ：

a) 規則 $U ::= \cdots RW$ が存在して $W \doteq S$ かつ U AS T である．

b) 規則 $U ::= \cdots RW\cdots$ が存在して $W \doteq ST\cdots$ である

c) 規則 $U ::= \cdots RWV\cdots$ が存在して $W \doteq S$ かつ V FIRST* T である．

5. $R \geqslant ST$ が成立つのは以下の条件の一つが成立し，しかもそのときに限られることを示せ：

a) 規則 $U ::= \cdots RS$ が存在して，U AS T である．

b) 規則 $U ::= \cdots RSV\cdots$ が存在して，V FIRST* T である．

6. 仮定 (6.2.4) のもとに，(1,2) (2,1) 順位文法で，任意の正準な文形式 $S_1S_2\cdots S_n$ の把手は，$S_{j-1} \lessdot S_jS_{j+1}$，$k=j, \cdots, i-2$ に対して $S_kS_{k+1} \leqq S_{k+2}$，$S_{i-1}S_i \gtrdot S_{i+1}$ を満たす系列 $S_j\cdots S_i$ の最も左側にある系列であることを証明せよ．

7. 定理 (6.2.5) を証明せよ．

8. 図 6.6 のプログラムと図 6.5 の表を用いて文法 (6.2.2) のつぎの各文を構文解析せよ：i + i + i，i * i + i，i + i * i，(i + i) * i．文でない系列 i i，i + * i の構文解析も試みよ．

9. 下記の文法に対する必要な全ての表を構成せよ．この文法は文法 (6.2.2) と同様であるが，演算子が左から右ではなくて右から左に評価されるところだけが違っている．

$$\text{E ::= T|T + E,}\quad \text{T ::= F|F * T,}\quad \text{F ::= (E)|i}$$

10. 問題 9 で作った表と文法を用いて下記の各文を構文解析せよ．

i + i + i，i * i + i

11. 文法 $A ::= (B)|dBe$，$B ::= c|cB$ は単純順位文法ではない．一

方，同じ言語に対する文法 $A ::= (B)|dBe$,　$B ::= c|Bc$ は単純順位文法である．このことを説明せよ．

6.3 限　定　文　脈

単純順位文法の問題点の一つに同一の右辺を持つ規則が2個以上あってはならないということがある．つぎの文法を例に考えてみよう．

(6.3.1)　$Z ::= aUa|bVb \qquad U ::= c \qquad V ::= c$

これは単純順位文法ではないが，文が与えられたとき c の前後の文脈をみて，c を U あるいは V のどちらに還元すべきかを云いあてることは容易である．文 aca では，c は U に還元，bcb では，c は V に還元されるだろう．限定文脈手法によれば，検知された把手の両側にある記号（複数でよい，記号の順位だけではなく，記号そのもの）を用いて，この把手を還元すべき記号を決定する助けとする．(1,2) (2,1) 順位技法でも，これがいくらか可能であった．

(1,1) 限定文脈の定義

$U ::= u$ を文法中の規則の一つとして，$Z \underset{c}{\overset{+}{\Rightarrow}} xTURy$ を仮定しよう．すると，正準な文形式を構文解析するときに，もし xTu がスタック中にあり，つぎの入力記号が R ならば，u を U に還元すべきであろう．任意の正準な文形式 $\cdots TuR \cdots$ の u が U に対する単純句であるということを TuR だけから云い当ててやりたい．すなわち，把手として可能なものと前後の文脈 $T \cdots R$ だけを使って，それが把手であることを云い，それを何に還元すべきかを教えてやりたいのである．もし，$Z \underset{c}{\overset{+}{\Rightarrow}} \cdots TuR \cdots$ であれば，把手としては，ほかにどんな可能性が生ずるのだろう．

まず第一に，われわれが行っている還元操作は，正準なものであるから，もし $\cdots Tu$ がスタック上に現れると，把手が u の最後の記号のまさしく左側にあり得るわけがないことがわかっている．これは，われわれの構文解析の性質のためである．把手の最後の記号が，スタックの上に置かれるやいなや，これは直ちに還元される．そこで，把手に対する可能性を以下のように列挙することができる．

(6.3.2)　1.　$\cdots Tu$

2. u; しかし $V \neq U$ では，これは V に還元されることもある．
3. u_2, ただし $u = u_1u_2$
4. $\cdots TuR \cdots$
5. $uR \cdots$
6. $u_2R \cdots$
7. 完全に u の右側にある場合

これらの各々の場合に対して，どのような正準な導出が，上のような把手を生成するのかを示すことができる，

(6.3.3) 1. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots VR \cdots$, ただし $V ::= \cdots Tu$

2. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots VTR \cdots$, ただし $V ::= u$ かつ $V \neq U$

3. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots Tu_1VR \cdots$, ただし $V ::= u_2$ かつ $u = u_1u_2$

4. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots V \cdots$, ただし $V ::= \cdots TuR \cdots$

5. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots TV \cdots$, ただし $V ::= uR \cdots$

6. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots Tu_1V \cdots$, ただし $V ::= u_2R \cdots$ かつ $u = u_1u_2$

7. $Z \underset{c}{\overset{+}{\Rightarrow}} \cdots TuV \cdots$, ただし $V \cdots \overset{+}{\Rightarrow} R \cdots$

上の7通りの場合がいずれも生じなければ，$\cdots TuR \cdots$ に対する唯一の可能な操作は，u を U に還元することである．

(6.3.4) **定義**　$Z \underset{c}{\overset{+}{\Rightarrow}} \cdots TUR \cdots$ となりうる全ての記号の対 T, R に対して，(6.3.3) の7通りの場合がいずれも生じなければ，規則 $U ::= u$ は（左から右の）(1,1) **限定文脈規則** (bounded context rule) である．■

(6.3.5) **定義**　文法に属する全ての規則が（左から右の）(1,1) 限定文脈規則であるならば，この文法を (1,1) **限定文脈文法** (bounded context grammar) という．■

ここで頭に“左から右の”ということばを付け加えているが，これは，右から左の構文解析に対する (1,1) 限定文脈も定義しよう思えば可能だからである．さらに，構文解析をどちらの方向からでも，あるいは中央から行うような限定文脈すら定義できよう．すなわち，どんな単純句でも，検知できようが，それは，把手ではない．

下記の文法は (1,1) 限定文脈文法ではない：

$$Z ::= abU|cbV \qquad U ::= c \qquad V ::= c$$

それは，$U ::= c$ と $V ::=c$ という規則で，(1,1) 限定文脈とはならないからである．記号列 $abc\#$ で，文脈 $bc\#$ から，右側にある c を還元するのに，$U ::= c$ あるいは $V ::=c$ のどちらの規則を用うべきかを云い当てることができないのである．

(1,1)　限定文脈認識法

この認識法で使う表は，ただ一つで，各行は 3 欄からなっている．第 1 欄は，**スタックの内容**で，全て Tu という形をしている．ただし T は，任意の記号か区切り記号 # であり(構文解析すべき文の終りに # を置く)，u は，規則の右辺である．第 2 欄は，対応する**右側の文脈** (right context) が入っている．これは，すべて，1 個の終端記号か区切り記号 # である．第3欄には，規則の番号が入っている．この表は，この認識アルゴリズムでつぎのように使用される．認識の各段で，表の第1欄がスタック上の要素と一致し，第2欄が，文中の今走査したばかりの記号と一致するような行を表の中で探す．もしそのような行が見付かれば，還元の操作を行うスタック上の把手を，第3欄で示す規則を用いて置換える．もし，そのような行が見付からなければ，還元は行われず，今走査したばかりの記号は，スタック上にのせられ，文のつぎの記号が走査される．

この表には，文脈 TuR で，u を i 番目の規則を用いて還元することができる全ての組合せ (Tu, R, i) が列挙してある．もし，表を引いて，一致するものがいつでも高々一つの行だけであるなら，文脈 $T\cdots R$ を知るだけで各還元操作が遂行でき，この文法は (1,1) 限定文脈文法である．**図6.7** は，下記の文法に対する表である．

(6.3.6)	(1)　$Z ::= V$	(4)　$V ::= bU$	(7)　$W ::= 0W1$
	(2)　$Z ::= aW$	(5)　$U ::= U0$	(8)　$W ::= 01$
	(3)　$V ::= V1$	(6)　$U ::= 0$	

この文法の言語は，二つの集合 $\{a0^n1^n|n>0\}$ と $\{b0^n1^m|n>0,\ m\geq 0\}$ の合併集合である．図6.7 をみると，還元を示すスタック上の Tu と，入力記号 R との可能な組合せが**全て**列挙してあることに注意しよう．

スタックの内容	右側の文脈	規則の番号
a 0 1	#	8
0 0 1	1	8
0 0 W 1	1	7
a 0 W 1	#	7
# a W	#	2
b 0	0	6
b 0	1	6
b U 0	0	5
b U 0	1	5
# b U	1	4
# V 1	1	3
# V 1	#	3
# V	#	1
スタック上に上の要素があり	走査された記号 R が上のものであれば	上の規則を用いて還元操作を行う

図 6.7 文法 (6.3.6) に対する限定文脈の表

1. $i := 1$, $k := 1$, $S(1) := \#$ を設定する.
2. もしスタックの上の要素が, 表の第1欄のいずれかと一致し, T_k が, 対応する第2欄と一致するならば, j を一致した行として, ステップ3.へ. さもなければステップ4. へ.
3. スタックに第 j 行の第 1 欄で示される Tu があるので, 第 j 行の第 3 欄によって示される規則を用いて, u を U に還元する. ステップ2.へ.
4. T_k が # に等しければ, ステップ6.へ.
5. $i := i+1$, $S(i) := T_k$, $k := k+1$ として, ステップ2. へ.
6. 終了. もし $i := 2$ かつ $S(i) := Z$ ならば, そしてそのときに限り, 文が認識された.

図 6.8 (1,1) 限定文脈認識アルゴリズム

この認識アルゴリズム自体は**図 6.8** にある. この図では, 例によって, スタックを S その指標を i, 解析すべき文を $T_1 \cdots T_n$ としてある. 文の区切り記号として $T_{n+1} = \#$ を付加し, スタックには, 始めに # をのせておく. カウンタ k は, 文を走査するために用いる.

(m, n) 限定文脈の定義

ちょうど，(1, 1) 順位文法を (m, n) 順位文法に拡張したように，(1, 1) 限定文脈も，(m, n) 限定文脈に拡張できる．実際には (m, n) 限定文脈を全面的に用いているコンパイラは存在しない．もっとも，一部に用いているものはいくつかある．したがって，ここでは，基本的なことを概観するだけにしよう．これまでに述べてきたこれ以外の認識法やこれから述べようとする認識法の全てによって受理される文法が適当な m, n に対して，(m, n) 限定文脈文法であるから，ここで，この方法を述べる．

(6.3.7) **定義**　規則 $U ::= u$ を考えよう．適当な x, y に対して $Z \overset{*}{\underset{c}{\Rightarrow}} xvUwy$ となるような記号列 v と w の全ての対に対して（ただし $|v| = m$, $|w| = n$ とする），u が，全ての正準な文形式 $\cdots vuw \cdots$ の把手であり，u が，記号 U に一意的に還元されるならば，この規則は (m, n) **限定文脈**である．■

この定義の意味はつぎのとおり：$U ::= u$ を限定文脈規則とし，さらに正準な導出 $Z \overset{*}{\underset{c}{\Rightarrow}} xvUwy \underset{c}{\Rightarrow} xvuwy$ が存在すると仮定する．すると，左から右，下から上へ向かう構文解析法ではスタック上に $\cdots vu$ があり，まだ処理されていない入力記号列の残部が $w\cdots$ のときはいつでも，この u が U に還元できる．ここでは，(1, 1) 限定文脈の場合にしたように，生じてはならない場合の列挙を行わなかったことに注意．これは読者に任せよう．

(6.3.8) **定義**　全ての規則が (m, n) 限定文脈であるならば，文法は (m, n) **限定文脈**であるという．■

(m, n) 限定文脈の認識アルゴリズムと自動作成機

(1, 1) 限定文脈の場合と同様に，三つの欄からなる表を使用する．第1欄は，**スタック上の内容** tu である．第2欄は，それに対応する**右側の文脈** v で，第3欄は，$\cdots tuv \cdots$ を $\cdots tUv \cdots$ に還元するのに用いる規則の番号である．認識法自体は，本質的には，(1, 1) のものと同じで，違いは，各ステップで比較しなければならない記号の個数が多いということである．

表の自動作成機は，読者が想像されるとおり，かなり複雑なアルゴリズムになる．第1ステップは，(1, 1) 限定文脈表を組立てることである．すなわち，全ての正準な導出 $Z \overset{*}{\underset{c}{\Rightarrow}} \cdots VUW \cdots \underset{c}{\Rightarrow} \cdots VuW \cdots$ に対して，これが表にエントリされていなければ，第1欄 Vu, 第2欄 W, 第3欄 $U ::= u$ という

規則の番号という行が表に加えられる（このような導出を全て調べることはできないと読者は思われよう．導出の数は無限にある．しかし，それぞれ規則 $U ::= u$ と記号 V, W に対して上のような導出が存在するかどうかを示すような方法で，文法を調べることが可能である）．

一たんこの表ができ上ると，表の各行で

V_1u_1	W	n
V_2u_2	W	m

の形式の対の有無を調べてみる．ここで V_2u_2 は，V_1u_1 の後部である（必ずしも真の後部ではない）．このような対が存在しなければ，この文法は，(1, 1) 限定文脈であり，この表が求めるものである．認識部の各ステップで，表の高々1行とスタックの内容が一致するので，この文法は (1, 1) 限定文脈である．

しかしながら，2行（またはそれ以上の行）が一致するような場合を仮定しよう．すると V_1u_1 がスタック上にあり，W が入力したばかりの記号とすると，一致したどの行をとっても還元ができる．したがって，表を変更して，より以上の文脈を——W の右側または，V_2u_2 の左側に——加えなければならない．例えば，左側に加える場合を考えよう．さらに規則 m を用いて $\cdots V_2u_2W\cdots$ を含む全ての正準な導出は，$Z \overset{*}{\underset{c}{\Rightarrow}} \cdots XV_2UW\cdots \underset{c}{\Rightarrow} \cdots XV_2u_2W\cdots$ という形式かまたは $Z \overset{*}{\underset{c}{\Rightarrow}} \cdots YV_2UW\cdots \underset{c}{\Rightarrow} \cdots YV_2u_2W\cdots$ という形式をしていると仮定しよう．すると，上の表の行をつぎのように変更することができる：

V_1u_1	W	n
XV_2u_2	W	m
YV_2u_2	W	m

これであいまいさが除去できるかもしれないし，できないかもしれない．もしできなければ，再び左側または右側に文脈を追加しなければならない．このように，衝突が含まれる行を一つづつ調べて，それを書き換えなければならないことに注意．残りの行はそのままで正しい．問題の一つは，左右どちらの文

脈を追加すべきかを知ることである．さらに，この文法が限定文脈であるのかないのか，衝突が全くない表が実際に得られるまで，わからないというのも問題である．

6.4 推　移　行　列

これまでに述べた方法の大部分で問題になるのは，還元を探すために，つねに規則や表の探索を行わねばならないことである．もちろん，探索時間を最小にするように表の構造を工夫することも可能である．ここで議論する技法は，大規模な行列で，時には，決定表とか推移行列と称せられる行列を用いて，この探索時間を最小化しようというものである．まず，演算子文法（OG）を仮定しよう．何となれば，この形式化は，推移行列を直観的に使用することに対応しているからである．つぎの演算子文法

(6.4.1)　〈プログラム〉 ::= 〈実行文〉
　　　　〈実行文〉　　::= IF 〈式〉 THEN 〈実行文〉
　　　　〈実行文〉　　::= 〈変数〉 := 〈式〉
　　　　〈式〉　　　　::= 〈式〉＋〈変数〉|〈変数〉
　　　　〈変数〉　　　::= i

を例にして，この技法を説明しよう．

まず推移行列をつぎのようにして作る．行の見出はスタックに現れる可能性のある規則の右辺の頭部（最後は終端記号で終っている）を表し，列の見出しは，文の区切り記号を含めて，終端記号を表している（図6.9**a**）．行列の要素には，サブルーチンの番号または先頭番地が入る．

(a)

	#	IF	THEN	:=	+	i
#		1				1
IF						
IF 〈式〉 THEN						
〈変数〉 :=						
〈式〉 +						
i						

(b)

	#	IF	THEN	:=	+	i
		1				1
	2		2	2	2	

図6.9　部分的に埋まった推移行列

認識部は，典型的なスタック S と，入力記号用変数 R を使用する．もっともスタック構造は，幾分ちがったものになっている．スタックの要素には，記号だけではなく，記号の系列も現れるようにする．ここで現れる系列は，規則の右辺の頭部（最後は終端記号で終っている）である．例えば，このスタックにある時点で

#IF 〈式〉 THEN IF 〈式〉 THEN 〈変数〉：＝

のような内容が入っていれば，それは

	〈変数〉	:=
IF	〈式〉	THEN
IF	〈式〉	THEN
	#	

のようになっているだろう．ここで行おうとしているのは，同時に還元しなければならないとわかっている記号は一しょにまとめておくということにすぎないことに注意．最後に，3番目の変数 U が必要である．この変数は，空であるか，最後の素句を還元したばかりの記号が代入されている．したがって，それまでに構文解析の済んだ記号列の一部が

#IF 〈式〉 THEN IF 〈式〉 THEN 〈変数〉：＝〈式〉

であるならば，スタックの中は上で示したようになっており，U には，〈式〉が入っていることになろう．

この認識部は，推移行列をつぎのようにして使用する．解析の各ステップでスタックの最上段の要素が，行列のいずれかの行に一致する．何となれば，両者は，ともに右辺の頭部（最後が終端記号で終る）を表しているからである．変数 R にある入力したばかりの終端記号が，行列の列を決定する．この両者で行列の要素，すなわち実行すべきサブルーチンの番号が定まる．このサブルーチンは，必要な還元を行ったり，R をスタックにのせて，つぎの入力記号をとってきたりする．

例えば，認識処理の最初は，スタックには#があり，U は空である．入力記号は，文法 (6.4.1) に従って，IF か i のいずれかである．これらの記号で右辺が始まっているので，これらをスタックにのせ先に進む．したがって，最

初に呼ぶサブルーチン（図6.9a 参照）は

1：IF U≠' ' THEN ERROR; i :=i+1; S(i) :=R SCAN

となる．ここで SCAN は，入力記号列からつぎの記号を走査して R にとってくるルーチンを意味する．U が空でないことを調べることについてはいずれ明らかになる．

i がスタックの最上段にあると仮定しよう．R にある記号はどんなものが有効なのか．i のつぎには，#，THEN，:=，＋ がつづくことが可能である．これらのいずれの場合にも i を〈変数〉に変えて，…〈変数〉…⇒…i… という導出を行いたいのである．サブルーチン2（図6.9b 参照）は，これを行うもので

2：IF U≠' ' THEN ERROR; i :=i−1; U := '〈変数〉'

となる．そこで，行列の要素には，それぞれ，入力してきた記号をスタックにのせるか，還元を行うかのいずれかを実行するサブルーチンの番号が入る．スタックを行う方は，それぞれのスタック要素によって，もう少し込入っている．文法（6.4.1）に対する完全な行列とサブルーチンを図6.10に示した．これらを用いて #IF i+i THEN i:=i# の構文解析を行ってみることを奨めよう．

推移行列の利点と欠点

この方法は，いくつかの初期の ALGOL コンパイラで用いられた．推移行列は，今ここで示したと同様な方法で，全て手で作成した．行列の各要素は可能な入力記号とそれに対応する行動を何にすべきかを言語から決定することによって，埋めていった．推移行列を使うことによって，コンパイラの書き手は，一時に，ある特別な構成要素に集中させられ，したがって，コンパイラ作成計画を，いくつかの，小部分に分割するのに役だった．この技法は，表引き操作が全く不要なので，非常に高速である．各サブルーチンは自分のなすべき仕事を精しく知っている．さらに，エラーの回復処理を組込むことが，極めて容易なことも，別の利点である．行列中のゼロ要素は，構文上の誤りに対応し，半分以上の要素が通常ゼロであるから，これらの誤りを処置するルーチンをたくさん書いて，何通りかの誤り回復法を持つことができる．これ以外の技法は全て，誤りの回復をいろいろな場合に分けるのが，実際には容易でないの

	#	IF	THEN	:=	+	i
#	5	1	0	6	0	1
IF	0	0	9	0	3	1
IF 〈式〉 THEN	7	1	0	6	0	1
〈変数〉 :=	8	0	0	0	3	1
〈式〉 +	4	0	4	0	4	1
i	2	0	2	2	2	0

```
4: IF U ≠'〈変数〉' THEN ERROR;
   i :=i−1; U:='〈式〉'.

5: IF U ≠ '〈プログラム〉' OR
      U ≠ '〈実行文〉' THEN ERROR;
   STOP.

6: IF U ≠ '〈変数〉' THEN ERROR;
   U:= ''; i :=i+1;
   S(i):='〈変数〉 :='; SCAN.

0: ERROR; STOP.

1: IF U ≠ '' THEN ERROR;
   i :=i+1; S(i):=R; SCAN.

2: IF U ≠ '' THEN ERROR;
   i := i − 1; U:='〈変数〉'

3: IF U ≠ '〈式〉' OR
      U ≠ '〈変数〉' THEN ERROR;
   i :=i+1; S(i):='〈式〉 +';
   U := '';    SCAN.

7: IF U ≠ '〈実行文〉'
   THEN ERROR;
   i := i − 1; U:='〈実行文〉'.

8: IF U ≠ '〈変数〉' OR
      U ≠ '〈式〉'
   THEN ERROR;
   i :=i−1; U:='〈実行文〉'.

9: IF U ≠ '〈変数〉' OR
      U ≠ '〈式〉'
   THEN ERROR;
   S(i):='IF 〈式〉 THEN';
   U := '';    SCAN.
```

図6.10 文法 (6.4.1) に対する推移行列とサブルーチン

で誤り回復のよい方法が工夫できないのである．この技法の欠点の一つは，記憶容量の問題である．IBM 7090 の典型的な ALGOL コンパイラでは 100 × 45 の行列と約 250 のサブルーチンを持っていた．

拡大型演算子文法の構成

これまでは，推移行列の働き方を直観的に議論してきた．ここでは，行列自体の構成を概観する．証明は全て演習問題として読者に任せることにする．形式的な接近法は，直観的な接近法とほぼ平行していることを心に留めてほしい．

まず，演算子順位法の場合のように，OG から始めよう．さらに，ある非終端記号から他の非終端記号の導出は一意的であると仮定しよう．すなわち，もし $U_i \stackrel{+}{\Rightarrow} U_j$ ならば，そのような導出は，ただ一つしか存在しない．

第一のステップは，この文法を，各規則の右辺に高々 2 個の記号しか現れないような等価な文法に，変換することである．このために，新しい非終端記号を導入する．この記号を**星付き非終端記号** (starred nonterminals; 略して

SNTS）と呼び，もとからある**星の付かない非終端記号**（unstarred nonterminals; 略して **UNTS**）と区別することにしよう．さらに，各 SNTS は，もとの OG 規則の右辺の頭部（最後が終端記号で終る）を表すように構成され，したがって先に述べた推移行列の行とそのまま対応する．この新しい文法を，**拡大型演算子文法**（augmented operator grammar; 略して **AOG**）と称する．これは必ずしも OG ではない．

この文法の構成は以下のとおりである．まず第1ステップを行いこれを適用できなくなるまで繰返す．つぎに第2ステップも同様に行う．最後にステップ 3a と 3b を適宜，適用できなくなるまで繰返す．以下の各ステップで，T は終端記号，U は非終端記号，U' は SNTS である．

1. もし規則 $U_1 ::= Ty_1$（y_1 は空であってよい）が存在し，それまでに $k-1$ 個の SNTS $U_1', U_2', \cdots, U_{k-1}'$ が導入されていれば新しい SNTS U_k' を作り，$U_i ::= Ty_i$（同じ T で始る全ての規則）を全て，$U_i ::= U_k'y_i$ で置換え，$U_k' ::= T$ という規則を追加する．
2. もし規則 $U_1 ::= UTy_1$ が存在して，$k-1$ 個の SNTS がそれまでに作り出されていれば，同じ UT で始る全ての規則 $U_i ::= UTy_i$ を $U_i ::= U_k'y_i$ で置換え，$U_k' ::= UT$ という規則を追加する．
3. (a) もし規則 $U_1 ::= U'Ty_1$ が存在し，$k-1$ 個の SNTS がそれまでに作り出されていれば，新しい SNTS U_k' を作り，$U_i ::= U'Ty_i$ を全て $U_i ::= U_k'y_i$ で置換え，新しい規則 $U_k' ::= U'T$ を追加する．

   (b) もし規則 $U_1 ::= U'UTy_1$ が存在し，$k-1$ 個の SNTS がそれまでに作り出されていれば，新しい SNTS U_k' を作り，$U_i ::= U'UTy_i$ を全て $U_i ::= U_k'y_i$ で置換え，新しい規則 $U_k' ::= U'UT$ を追加する．

上の各ステップの終了後にできている規則の形式に注意して，つぎの補題を証明してもらう（演習問題 1 参照）．

(6.4.2)　**補題**　　AOG では全ての規則はつぎの7通りのいずれかの形式をしている：

$$U_1 ::= U_2,\quad U_1 ::= U',\quad U_1 ::= U'U_2$$
$$U' ::= T,\quad U' ::= UT,\quad U_2' ::= U_1'T,\quad U_2' ::= U_1'UT \blacksquare$$

つぎは，文法 (6.4.1) から AOG を作ってほしい（演習問題 2 参照）．そし

てつぎの二つの補題と定理を証明してもらおう.

(6.4.3) **補題** OG の各規則 $U::=y$ (ここで y は非終端記号ではない) に対して, $y=y_0y_1\cdots y_n$ であるような AOG の規則の集合 $U_1'::=y_0$, $U_2'::=U_1'y_1$, $\cdots$, $U_n'::=U_{n-1}'y_{n-1}$, $U::=U_n'y_n$ が一意的に存在する.■

(6.4.4) **補題** OG のそれぞれ相異なる文形式 S の構文の木に対してつぎのようにして, AOG の S に対応する相異なる構文の木を構成することができる. すなわち規則 $U_1::=y$ を適用することに対応する枝をそれぞれ, 補題(6.4.3) に述べた規則群を適用することに対応する枝の集合で置換える.■

(6.4.5) **定理** もし, ある AOG があいまいでなければ, それに対応する OG もあいまいでない.■

補題 (6.4.4) による木の構成の説明として, 図6.11**a** の木の枝を, 図6.11**b** の部分木で置きかえる. もとになる文法は (6.4.1) である. 図中で ⟨<u>if</u>⟩ と ⟨<u>if—then</u>⟩ が SNTS である. 読者は自分で, AOG に従って行う上向き構文解析が, 本質的には, OG に従うものと同じであることを確かめてほしい. 違いといえば, 中間的な導出がいくつか導入されているだけである.

AOG における構文解析

上向き構文解析法は, 各ステップで最も左側の **AOG の素句**を認識する. ここで, AOG の素句とは, その中にこれ以外に AOG の素句を含まず, 少くとも1個の終端記号か, あるいは SNTS を含むような句と定義する. この定義を (6.1.7) と対比してみよ. するとつぎのいずれか一方が云える. すなわち V が SNTS または UNTS のとき $V::=y$ であるような文形式 xVz

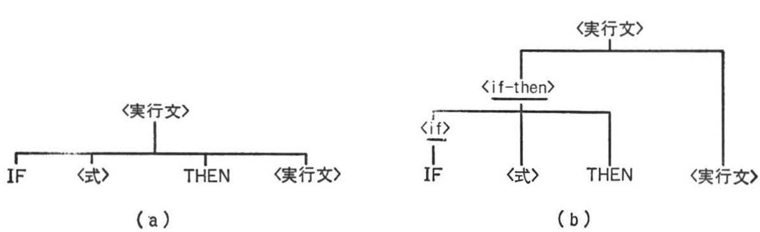


図6.11 OG と AOG の対応する枝

が存在するならば，y は文形式 xyz の AOG の素句であるか，あるいは $V ::= y_1$，$y_1 \overset{*}{\Rightarrow} y$ で，かつ $y_1 \overset{*}{\Rightarrow} y$ 中に現れる全ての直接導出が UNTS U_i および U_j に対して，規則 $U_i ::= U_j$ を適用したものになっているかである．

補題 (6.4.2) から，AOG の素句はつぎの6通りの形式のうちいずれかでなければならない．

(6.4.6)　U', $U'T$, T, $U'U$, UT, $U'UT$

したがってつぎの定理が成立つ（証明は演習問題 6)．

(6.4.7)　**定理**　AOG の文の構文解析列の各直接導出（最も左側の AOG の素句を還元するもの）は，つぎの6通りの形式のいずれかをとる：

$$u'Ut \Rightarrow u'U't,\ u'U_1't \Rightarrow u'U_2'Tt,\ u'U_1't \Rightarrow u'Tt$$
$$u'U_1t \Rightarrow u'U'U_2t,\ u'U_1't \Rightarrow u'U_2'UTt,\ u'U't \Rightarrow u'UTt$$

ここで u' は SNTS の記号列（空であってもよい），t は終端記号からなる記号列，U, U_1, U_2 は UNTS, U', U_1', U_2' は SNTS である．したがって，構文解析列中の任意の文形式の形は $u't$ または $u'Ut$ である．■

この構文解析アルゴリズムは，文形式 $u'(U)t$ 中の部分記号列 u' を納めるためにスタック S を用いる．変数 U は記号 U が存在すれば，それを保持する．存在しなければ空である．行列はタテの欄を各 SNTS，横の行を各終端記号とする．各 SNTS はタテの欄に対応するので，それらを，スタック中では，その行の番号で表すことができることに注意しよう．

構文解析の各ステップで，スタックの最上段にある記号（SNTS の一つ）とつぎに入ってくる記号で，行列の要素が決定される．これは実行すべきサブルーチンの番号である．このサブルーチンは，変数 U の中を調べて，適宜の処置をとり，6通りの可能な還元のいずれかを実行する．

次節でつぎのような十分条件を与えよう，すなわち，この条件が満たされれば，構文解析の各ステップで，スタック上の記号と，入ってくる終端記号と，変数 U の値からだけで，還元が可能となる十分条件を与えよう．

あいまいでない AOG の十分条件

さて，AOG があいまいでないためのいくつかの条件を与えよう．さらに，

これらの条件によって，ある構文解析の各ステップで最も左側の AOG の素句を検出しそれを還元すべき記号を探すことが可能になる．全ての文形式は $u'[U]t$ という形式であることがわかっている（ここで [ ] は記号 U が存在してもしなくてもよいことを示している）．これらの条件によって，u' の最後の記号と，t の最初の記号と変数 U の値とから，還元があるとすればどんな還元を行うべきかを決定できるであろう．

(6.4.8) **定義**　AP(S) とは，記号 S の前に置き得る記号（許容前置子，演習問題 6.2 参照）の集合である：$\mathrm{AP}(S) := \{S_1 | \cdots S_1 S \cdots$ は文形式である$\}$. ■

条件 1. U' を SNTS，T を終端記号とするとき，U'，T の対のそれぞれに対して，つぎの三つの表明の高々一つが成立する．

(6.4.9)　規則 $U ::= U'$ が存在する．ただし U は AP(T) に属する．

(6.4.10)　規則 $U_1' ::= U'T$ が存在する．

(6.4.11)　規則 $U_1' ::= T$ が存在する．ただし U' は AP(U_1') に属する．

さらに，もし (6.4.9) が成立すれば，そのような UNTS U はただ一つ存在する．■

条件 2. U' は SNTS，U は UNTS，T は終端記号とする．3-組 U'，U，T に対して，つぎの三つの言明の高々一つが成立する．

(6.4.12)　規則 $U_1 ::= U'U_2$ が存在する．ただし U_1 は AP(T) に属し $U_2 \overset{*}{\Rightarrow} U$.

(6.4.13)　規則 $U_1' ::= U'U_2T$ が存在する．ただし$U_2 \overset{*}{\Rightarrow} U$.

(6.4.14)　規則 $U_1' ::= U_2T$ が存在する．ただし U' は AP(U_1') に属し $U_2 \overset{*}{\Rightarrow} U$.

さらに，もし (6.4.12) が成立すれば，U_1 と U_2 は共に一意的である．一方，もし (6.4.13) または (6.4.14) が成立すれば，U_2 は一意的である．■

以下の定理が成立つ（証明は演習問題 7 に任せる）．

(6.4.15) **定理**　条件 1 および 2 を満たす AOG はあいまいでない．■

演習問題 6.4

1. 補題 (6.4.2) を証明せよ.

2. 文法 (6.4.1) に対する AOG を構成せよ. まず, 第1段は, 2番目の規則を U_1' ::= IF と 〈実行文〉 ::= U_1'〈式〉THEN〈実行文〉 の2個の規則で置換え, 最後の規則を U_2' ::=i と 〈変数〉 ::= U_2' で置換えよ.

3. 補題 (6.4.3) を証明せよ (この補題は構成法から容易に導ける).

4. 補題 (6.4.4) を証明せよ.

5. 定理 (6.4.5) を証明せよ. **ヒント**: AOG の文形式に対する構文の木が一意的であれば, OG における, これに対応する木も一意的でなければならない.

6. 定理 (6.4.7) を証明せよ. **ヒント**: AOG の素句がとり得る6通りの形式とそれに対応する還元の形式に注意.

7. 定理 (6.4.15) を証明せよ.

6.5 参考文献概説

Floyd (63) は演算子文法という考え方を形式化した. これは, はじめの頃のコンパイラで, 直観的な方法で使用されていたものである. Allard ほか (64), Galler と Perlis (67), Gries ほか (65) も参照せよ. Wirth と Weber の単純順位文法は, ALGOL W コンパイラ (Bauer, Becker と Graham (68) 参照) で用いられている. Wirth と Weber は (m, n) 順位文法も導入しているが, 彼らの定義はよくない. (1, 2) (2, 1) 順位法への拡張は McKeeman (66) によるものである. これはさらに整理されて McKeeman ほか (70) による XPL コンパイラ記述システムで用いられている. Colmeraur (70) は, 演算子順位法と単純順位法をともに包含する順位文法を定義した. 簡単にいえば, コンパイラの書き手は, 自分の演算子を選ぶことができる (これには, 二, 三制限がある. 例えば, 全ての終端記号は演算子であるなど). もし, 書き手が終端記号群を選定すると, 彼には演算子順位技法が直ちに得られている. 全ての記号を演算子に選べば, 単純順位技法が得られる.

何人かの人達が, 単純順位法をより使い易くするために拡張を試みてきた. 例えば, Ichbiah と Morse (70) は, “弱い順位” 関係を定義して, 関係に代

えてパターンの一致を調べてスタック中の把手の頭部を見出すために用いた．これは (1, 2) (2, 1) 順位法の場合にわれわれが行ったことから直接出てくる．Zimmer (70) は，右辺の一意性についての条件をゆるめ還元を行うのに使用すべき規則を決定するのに文脈を検査させる．これは (1, 1) 限定文脈法へ接近する始まりである．

(1, 1) 限定文脈は Paul (62) に源を発し，最初の論文は Eickel ほか (63) であった．Floyd (64a) と Irons (64) はともに，これを，(m, n) 限定文脈に拡張したが，両者の定義は，同じではなかった．本書では Floyd の形式化に従っている．これら 2 論文は 1963 年 8 月に開かれた，機械的言語構造の作業部会で発表されたものである．CACM 1964 年 2 月号は，この会議の報告集となっており，一読の価値がある．左から右の (m, n) 限定文脈認識アルゴリズム用の表を構成する方法を最初に示した論文は Eickel (64) のものであった．この論文は，読みこなすのが困難であるばかりでなく，入手するのも困難である．Loeckx (69) はこの表を構成する新しいアルゴリズムを導入した．こちらの論文はずっとわかりやすい．

これらの左から右，下から上への認識法の最終版が Kunth (65) の LR(k) 文法および言語である．Knuth は LR (k) 文法が与えられたとき，それに対する認識法を構成する方法を示している．これは，とるべき還元を決定するためにスタック中の全ての記号と，右側に k 個の記号を見るものである．実際には，各ステップ毎に，スタックの中を全て見る必要はない．スタック中の記号にはそれぞれ，スタックの残りのものに関する情報を提供する“状態”を付随しておいて，決定が十分下せるようにしておく．

LR(k)，あるいはそれを一部制限したもの (DeRemer (70) による) は能率の点でコンパイラで実用に供しうることを云っておくべきであろう．これはうまく作成すれば，(1, 2) (2, 1) 順位技法よりも，ずっと速く，少ない記憶容量で実現しうることを Horning (70) が指摘している．

推移行列法は，最初 Samelson と Bauer (60) によって発表され，ALCOR グループによって書かれたいくつかの ALGOL コンパイラで用いられた (Gries ほか (65) 参照)，NELIAC 作成グループも推移行列を使用したが (Halstead 参照)，スタックなしで，CO-NO (Current Operator and Next Operator) 表という．この方法は Gries (68) によって形式化された．

ここで，このほかの構文認識の過程を形式化したものについても言及しておくべきであろう．もっとも，全てが上向き認識法というわけではない．Lynch

(68) は ICOR 文法を定義しその構文解析アルゴリズムを与えた．Domolki のアルゴリズムは Hext と Roberts (70) にある．Gilbert は文脈依存文法に，**選択関数** (selection function) を加えた．この関数は（現在の文形式に基いて）還元を行うためにどの規則を用いるべきかを示すものである．したがってこの文法は，分析的というよりはむしろ合成的なものである．この文法は自動作成機で処理する必要もなくて，構文解析アルゴリズムとして使用可能である．

Tixier (67) は，1個の非終端記号に対する規則の集合がそれぞれ，その非終端記号に対する記号列の集合を定義する正規表現であると考えた．それぞれの集合の記号を認識する fa を構成することができて，問題は先ず，ある fa から他の fa に切換える時点を知り，つぎに，切換え先をどの fa にするかを知ることである．例えば，もし $V ::= Ax|By$ があれば，集合 A からの記号列を認識する fa に切換えるか B に対する fa に切換えるかということである．Tixier は，この問題を限られた文脈の中で解決するために，彼の自動作成機によって受理される文法に適当な制限を課している．

これまでに述べた認識法は，全て出力を出さない．すなわち，これらは，文法の文を認識はするが，文を，実際に別の形に翻訳はしていない．これを行うのは，12章で導入する，いわゆる意味付けルーチンと称せられるものである．Lewis と Stearns (68) は，構文向き変換機を定義した．これは，文を認識するだけでなく何らかの形式に翻訳する．モデルは形式的なものでコンパイラの作成で実際使うのには，まだ十分一般的なものとなっていない．

Stearns と Lewis (68) と Whitney は独立に，文脈自由文法を拡張して，構文解析の各段で，ある種の表を読み書きするようにした．目的は，表中に名前を探し，その属性を検査したり変更したりする概念の形式化にある．したがって，言語の定義の大半が形式化でき，コンパイラを生成するのが容易になる．

第 7 章

生　成　言　語

生成言語 (production language; 略して **PL**) とは，構文認識アルゴリズムを記述するためのプログラム用言語である．このプログラムは，主として**生成規則** (productions) の系列からなっている．これは，文法の書換え規則という普通に使う意味とまぎらわしいので，この言葉をここで使うのは不幸なことである．ここでいう生成規則は，むしろ**還元規則** (reductions) と呼んだ方がよさそうである．何故ならば，これらの規則は，大抵の場合（必ずしもいつもというわけではない）文を文法の出発記号に還元するために使用しているからである．新しい術語を設けるよりはこれまで本書で一貫して用いてきた**生成規則**という言葉をそのまま使うことにする．

PL は，普通，トランスレータ作成システム（アセンブラやコンパイラのようなプログラムを作成するために設計されたシステム）に埋込まれる．PL のプログラムはまず 3.4 節に述べた走査部の定義によく似たものから始まる．したがって PL のプログラムは，原始言語記号——予約語，名前，そのほか——の内部表現を扱い，実際の文字は取扱わない．

PL の記法は，作成者ごとに異なっており，ここで提示するものが，唯一つのものだとか最善のものだとか考えてもらっては困る．ここでの主たる興味は，PL プログラムが行う内容にあり，PL プログラムをカード上にどうパンチするかということではない．

7.1 言　　語

生成規則

PL プログラムは，SYN と称する LIFO スタックを自動的に使用しており，これは，これまでに述べてきたほかの認識法の場合と全く同様である．ス

タックには，原始言語の記号か，非終端記号に似た性格のものが入る．プログラム自体は，もともと，生成規則の系列であり，それぞれは

[〈名札〉:] 〈記号〉 {〈記号〉} [→{〈記号〉}] {〈操作〉}
　　　　　└───左辺───┘ └───右辺───┘

のような形式である．〈記号〉とは，原始言語の記号そのものであるか，非終端記号と呼ばれる名前（多分かっこ〈と〉で括られている）であり，〈名札〉とは，普通に用いる任意の名前である．上で注意したように〈名札〉がある場合その直後につづく〈記号〉（複数個でよい）が生成規則の**左辺**を形成しており，**右辺**の方は，矢印 → とそれに続く〈記号〉（複数個でよい）からなっている．

1個の生成規則は，あるパターン一致とスタックの変換を示しており，それは以下のとおりである．PL プログラムの実行中には，**只今の** (current) 生成規則がつねに存在する．生成規則が只今になると，その左辺にある〈記号〉を LIFO スタック SYN の最上段にある〈記号〉と比較する．記号に1個でも一致しないものがあれば，生成規則の系列の中のつぎに続く規則が只今になり，再びこれと比較を行う．これを一致がとれるまでつづける．スタック上の一致した〈記号〉（複数個ありうる）は，生成規則の右辺にある〈記号〉（複数個ありうる）と置換えられる．（右辺に全くなにもない場合には，スタック上では，何の変更も行われない．）〈操作〉が存在すれば，これは，書かれた順に実行される．ともあれ，この〈操作〉（複数個でよい）の系列で，原始プログラムの記号を走査し，スタックに置き，つぎの只今の生成規則を示すようになっている．

(7.1.1)　**例**．つぎの生成規則

COND : IF 〈E〉 THEN → 〈ICL〉 SCAN GO THENPART

が，只今の生成規則であるとしよう．スタックの最上段，2段目，3段目の要素が，それぞれ THEN, 〈E〉, IF であれば，以下のようになる：

1. スタックの上から3段に入っている要素を SYN から削除する．
2. 記号〈ICL〉をスタックにのせる．
3. 以下の2操作を実行する：
   a) SCAN — つぎの原始言語記号を走査し，スタックにのせる．
   b) GO THENPART — THENPART と名札の付いた生成規則が只今の規則になる．

これを書換え規則の言葉に直すと，規則〈ICL〉::= IF〈E〉THEN を用いて，IF〈E〉THEN を〈ICL〉に還元したことになる.■

図7.1 のプログラムには，生成規則の例がもっとたくさん挙げてある．後で見ることにしよう．これは，構文規則 (7.1.2) をもった算術式用の認識アルゴリズムである.

超言語記号

生成規則の中で超言語記号 ANY が〈記号〉として，しばしば用いられている．この意味は，普通の非終端記号とは違う．これが規則の左辺に現れると，対応するスタック上の記号と**つねに**一致することにする.

右辺に ANY があれば，少なくともそれと同数の ANY が左辺になればならない．右辺にある〈記号〉はいずれもスタック上に置かれることを考えると，ここで，ANY に対してつぎのような意味を与える．すなわち，右から左に数えて，右辺にある i 番目の ANY が，左辺の中で i 番目の ANY と一致したスタック上の只今の記号である．これをはっきりさせるために例を二つ示そう.

只今の生成規則が

T＊F ANY → T ANY GO LE

で，スタックには，T＊F＋ がのっているとしよう．するとスタックの内容は左辺と一致する．右辺の ANY は記号 ＋ であり，これは左辺で ANY と一致をみたものであり，スタックは T＋ に置換わる．つぎに生成規則 LE が只今の生成規則になる.

もし，スタックに W X Y Z がのっており，只今の生成規則が ANY Y ANY → ANY U ANY V であれば，スタックは，書換わって W X U Z V となる.

ほかに超言語記号として，I：走査部によって検出された任意の名前と一致する，N：任意の整数と一致する，の二つがある.

操　　作

左辺がスタック上の記号と一致し，スタックの変換が終了すると，〈操作〉が実行される．非常に重要な〈操作〉に SCAN と GO の二つがある．これは，

ST:	#			SCAN	最初の記号をスタックにのせる.
LF:	I	→	F	SCAN GO LT	F になる最初の記号(I, N または()
	N	→	F	SCAN GO LT	を調べてそれに応じた処置をする.
	(			SCAN GO LF	
	ANY			HALT 2	エラー, 停止.
LT:	T * F ANY	→	T ANY	GO LT1	還元 T::=T*F
	F ANY	→	T ANY		還元 T::=F
LT1:	T *			SCAN GO LF	*の検査.もし*ならつぎのFをとる.
LE:	E + T ANY	→	E ANY	GO LE1	この三つの生成規則は, 還元 E::=
	E − T ANY	→	E ANY	GO LE1	E+T, E::=E−T, E::=T に対応.
	T ANY	→	E ANY		
LE1:	E +			SCAN GO LF	E が現れたときの実行の方法を与え
	E −			SCAN GO LF	ている.
	(E)	→	F	SCAN GO LT	還元 F::=(E)
	# E #	→	# Z #	HALT 0	還元 Z::=E, 停止.
	ANY			HALT 3	エラー, 停止.

図7.1　文法 7.1.2 のための PL プログラム

もうすでに示した. 〈操作〉 SCAN を実行すると, 走査部が呼出される. 走査部は, つぎの原始プログラムの記号をとってきて, それを構文スタックの最上段にのせて, 戻ってくる. 〈操作〉 GO 〈名札〉 を実行すると 〈名札〉 と名付けられた生成規則が, 只今となり, スタック上の要素との比較が開始される.

CALL, RETURN, EXEC などの 〈操作〉 は, 適宜説明しよう. このほか便利な 〈操作〉 には

ERROR 〈整数〉 ,　HALT 〈整数〉

などがある. 前者は標準の出力装置の上に "ERROR 〈整数〉" とメッセージを印刷するもので, 〈整数〉 は, 整数である. このあと認識アルゴリズムは停止する.

例

図7.1 は, 文法 (7.1.2) の文を認識する生成規則を表わしている. プログラム中で, 走査部が, 名前を I, 整数を N と認識することを, 再び仮定している. プログラムが実行を開始すると, このシステムは, 区切り記号 # をスタック上に置く. さらに文末にも # が現れるものとしよう.

(7.1.2)　Z ::= E

$$
\begin{aligned}
E &::= T|E+T|E-T\\
T &::= F|T*F\\
F &::= I|N|(E)\\
I &::= \text{英字}\ \{\text{英字}|\text{数字}\}\\
N &::= \text{数字}\ \{\text{数字}\}
\end{aligned}
$$

図7.1 を用いて文の構文解析を行う前に，このプログラムを少し議論しておこう．LF＋2 にある生成規則が，右辺のない生成規則の例である．したがって，ここで一致がとれてもスタックは変更されない．LT＋1 の生成規則には，操作 が全くないことに注意しよう．もし，生成規則に GO 操作 がないならば，全ての操作の実行が終った後に，つぎの行にある生成規則が只今の生成規則になる（今の場合は LT1）．

生成規則はつぎのようにして記述されたものである．生成規則 ST は，つねに一致するので，つぎは文の第 1 番目の記号を走査する．生成規則 LF から LF＋2 は，因子 F を始める記号を検査する．もし存在しなければ，生成規則 LF＋3 によって，エラーの印刷が行われる．もし，因子が I か N であれば，F への還元が行われ，つぎの記号の走査へ向い，生成規則 LT が只今になる．もし因子がかっこで始まっていれば，生成規則 LF＋2 によって，走査が行われ，かっこの中の算術式の最初の記号をとってきて再び LF に戻る．

2個の生成規則 LT と LT＋1 の順序は重要である．ここでは，スタック上につねに F ANY がのっており，生成規則 LT＋1 は，いつも F をに T 書換える（書換え規則 T ::＝ F）．しかしながら，これを行う前に，T ::＝ T＊F という規則が適用できないかどうか調べてみる必要がある．上のいずれかの還元が一たん行われると，＊がスタック上にあるかどうか調べる．もしあれば，つぎの記号を走査し，F に対する句を見付けるために LF に戻ることによって T ::＝ T＊F という還元を後に行うことに備えておく．もし，＊がそこになければ，E への還元を調べて適当な還元を行う．ここでまた，生成規則 LE から LE＋2 の順序が重要である．

最後に実行される生成規則は LE1＋3 である，これは，文の最後に区切り記号 # があることを確認して，E を Z に還元する．

図7.2 は文（A＋B）＊C の構文解析を示したものである．各行は，只今の生成規則，その時点でのスタックの内容，つぎの一致を生成する生成規則，つ

只今の生成規則	スタックの内容	一致した生成規則	つぎの生成規則
ST	#	ST	LF
LF	# (	LF+2	LF
LF	# (I	LF	LT
LT	# (F +	LT+1	LT1
LT1	# (T +	LE+2	LE1
LE1	# (E +	LE1	LF
LF	# (E + I	LF	LT
LT	# (E + F)	LT+1	LT1
LT1	# (E + T)	LE	LE1
LE1	# (E)	LE1+2	LT
LT	#F *	LT+1	LT1
LT1	#T *	LT1	LF
LF	#T *I	LF	LT
LT	#T *F #	LT	LT1
LT1	#T #	LE+2	LE1
LE1	#E #	LE1+3	HALT 0
	#Z #		

図7.2　図7.1 に従った (A + B) * C の構文解析

ぎの只今の生成規則の順になっている.

クラスの名

図7.1 の生成規則 LE と LE + 1 は + と − を除いて全く同じである. これを縮約するために, 全ての生成規則に先立って

```
CLASS PM + − ;
```

と宣言し, 2個の生成規則をただ一つの生成規則

```
LE : E PM T ANY → E ANY GO LE1
```

で置換える. 一般に, 宣言

CLASS 〈クラスの名〉〈記号〉{〈記号〉};

によって,〈記号〉が,〈クラスの名〉と結び付けられる. この名前は, 普通に使う名前でよい. クラスの名は, 左辺に用いられると, 宣言中に現れる記号とは, どれとでも一致する.

もし〈クラスの名〉が生成規則の右辺に現れれば, 右辺にある〈クラスの

名〉と少くとも同数の〈クラスの名〉が左辺に現れなければならない．右辺に現れる〈クラスの名〉の意味は，ANY と同様である．右から数えて，右辺の i 番目に現われた〈クラスの名〉は，現在スタックの上にのっている記号でそれは左辺の i 番目に現れる〈クラスの名〉と一致する．例えば，上の宣言を用いて，スタックに ＋ －A＋ があり，只今の生成規則が

PM A PM → B C PM PM

であればスタックは ＋B C－ ＋ と変化する．上の生成規則は，4 個の生成規則

＋A＋ → B C＋ ＋
＋A－ → B C＋ －
－A＋ → B C－ ＋
－A－ → B C－ －

と同等である．

7.3 節で，いわゆる**意味付けルーチン番号**とクラスの中の記号とを結びつける利点を議論することにしている．ルーチン番号 20 を＋と，2 を－と結びつけるためには，上の PM の宣言を書換えて

CLASSNO PM ＋ 20 － 2;

のようにする．このようなクラスの名は，前と同じように使用することができる．これ以外の使用法については 7.3 節で説明しよう．この新しい宣言の形式は

CLASSNO 〈クラスの名〉〈記号〉〈整数〉{〈記号〉〈整数〉}

である．クラスの名を使用しても，PL の機能が増強されるものではない．クラスの名を用いてできることは，それがなくとも可能である．しかしながら，クラスの名は，PL によるプログラム書きを，より簡単にし，さらに，プログラムの大きさを小さくするのに役立つ．

原始言語と PL 記号との重複

もし → が原始言語の記号であるとすると，

(7.1.3)　　ANY → ANY → B

という生成規則はどんな意味になるのだろうか．これは，もちろん，あいまいなものである．あいまいさを解決するためには（そしてそのほか起り得る事態を解決するためには），原始言語記号に ： → ＄ ； ＃ などの文字，あるいは CLASS, CLASSNO, I, N, ANY, CALL, ERROR, EXEC, GO などの PL 記号のどれか一つでも含むような場合には，頭に ＄ 印をつけなければならないという約束を採用する．＄ 印は，したがって，それに続く（空白文字が現れるまで）の文字列が，原始言語記号の1個分になることを示す拡張文字となっている．

この記法を用いて，生成規則 (7.1.3) に対する起りうる3通りの異なった解釈を書下してみるとつぎのようになる：

```
ANY    →  ANY $→  B
ANY $→  ANY $→  B
ANY $→  ANY    →  B
```

つぎの例は，生成規則 LABE : I ＄ : ＄＄ →〈名札〉は，スタックの最上段に "＄"，2段目に "："3段目に名前がある場合に一致がとれ，これら3個の記号を，〈名札〉という1個の記号と置換えるであろう．

構文のまとめ

PL プログラムの構文は以下のようにまとめられる：

〈プログラム〉 ::= {〈宣言〉} {〈生成規則〉}
〈宣言〉 ::= CLASS 〈クラスの名〉〈記号〉{〈記号〉}
　　　　　|CLASSNO 〈クラスの名〉〈記号〉〈整数〉
　　　　　{〈記号〉〈整数〉}
〈生成規則〉 ::= [〈名札〉:]〈左辺〉[〈右辺〉] {〈操作〉}
〈左辺〉 ::= 〈記号〉{〈記号〉}
〈右辺〉 ::= → {〈記号〉}

ここで

1. 〈名札〉と〈クラスの名〉は，それぞれ普通に用いる一意的な名前であって，原始言語の記号でも PL プログラムの記号でもない．
2. 〈記号〉とはそれぞれ
   a) 起言語記号 I, N, ANY のいずれかであるか；

b) 一意的な，普通に使う名前であって，おそらく，かっこ〈 〉に括られており，PL の記号でも原始言語の記号でもなく；

c) 原始言語記号，(ただし：→ $; # あるいは PL の記号のいずれをも含まない) であるか；

d) $ を先頭にした空白を含まない任意の文字系列 ；である．

3. 〈整数〉とは 0 から 9 までの数字からなる空でない系列である．

4. PL 記号とは，ANY, CALL, CLASS, CLASSNO, EXEC, GO, I, N である．

5. 〈操作〉とその意味は下記のとおり：

a) CALL〈名札〉.〈名札〉と名付けられた生成規則から始めて，生成規則を実行し，RETURN という操作を実行するまで続ける．そしてこれにつづく〈操作〉または生成規則を実行する．したがって，これはまさにサブルーチン呼出しであり，7.2 節で説明する．

b) ERROR〈整数〉. "ERROR〈整数〉" と印刷する．

c) EXEC〈整数〉.〈整数〉の番号がついた意味付けルーチンを実行する．実行が終ると，このつぎの〈操作〉の実行を続ける．(これは 7.3 節で説明する.)

d) EXEC〈クラスの名〉.〈クラスの名〉は，生成規則の左辺に現れなければならない．この宣言で，意味付けルーチンと記号とを結び付ける．左辺の〈クラスの名〉と一致したスタック上の記号と結び付けられている意味付けルーチンが実行される．

e) GO〈名札〉.〈名札〉と名付けられた生成規則を只今の生成規則とし，再び比較を開始する．この操作のあとに書いてある操作は，決して実行されない．

f) HALT〈整数〉. "HALT〈整数〉" を印刷し，PL プログラムの実行をやめる．

g) RETURN. 最後に実行された CALL のつぎに戻る (操作 a を参照).

h) SCAN. 走査部を呼出し，つぎの原始プログラム記号を組立ててそれをスタックの上にのせる．もはや，記号が存在しなければ，区切り記号 # がスタックにのせられる．

演習問題 7.1

1. 文法 (7.1.2) の下記のような文を，図7.1 の PL プログラムを（手で）実行することによって構文解析せよ．走査部は名前と整数を構文解析するものと仮定する： (a) A*B　(b) (A)　(c) A+B+1　(d) A*B*1　(e) A+(B+1)　(f) A*(B+1)

2. 下記の 6 個の生成規則を，〈クラスの名〉を利用して書換え，2 個の生成規則に減らせ．

```
X Y → Y,     X Z → Z,     X ANY → X,
W Y → Y,     W Z → Z,     W ANY → W.
```

3. 下記の生成規則はそれぞれどこが間違っているか．

   a) E + T → ANY
   b) L : L1 : E + T → E SCAN GO Y
   c) L : I : → 〈名札〉 SCAN GO STATE

4. 文法 $G[Z]$： Z ::= + |Z + |Z − の文を認識する PL プログラムを書け．

7.2 PL の使用

PL は，プログラム用言語であって，これまでに調べてきたこれ以外の構文解析部の大部分のように機械的に認識部を構成する自動作成機はないので，ずっと融通性がある．構文解析に際しては，形式的な文法に正確に従う必要はなく，能率のよいプログラム作りを試みることができる．さらに，構文を意味の都合に合わせるような融通性も持っている．あとの問題は，12 章と 13 章で議論する．この節では，PL でプログラムする上での二，三のヒントを与えたいと思う．まず，(m, n) 限定文脈認識法を PL でプログラムする方法を示そう．つぎに，4.3 節で議論した，再帰的下向き法をプログラムする方法を示そう．最後に，二，三のプログラム技法を説明するために簡単な例を示す．

PL の使用による上向き構文解析

ここでは，6.3 節の知識を必要とする．第 6 章の図6.7 は，文法 (6.3.6) に対する (1, 1) 限定文脈上向き認識法のための表であった．表の各行は，パターンの一致とスタック上の変換を示していた．例えば，第 1 行は，もし "a

0 1″ がスタック上にあり，# が入力してきた記号であれば，スタック上を ″a W″ に変えよという意味である．

記法を別にして，PL プログラムとこの表との主たる違いは，表は，アルゴリズムでもプログラムでもないことである．この表は，図 6.8 のアルゴリズムにとってはデータにすぎない．PL の生成規則は，この変換が終って何をなすべきかを指示している．ところが図 6.7 でのどの行も何も指示していない．

図 6.7 の表と図 6.8 のアルゴリズムを結合して，本質的には，同じ認識法となる PL プログラムを構成することができる．これを行っているのが**図 7.3**である．このプログラム中，右側の文脈 R は，つねにスタックの最上段に現れる．それは，こうすることが PL によって原始言語記号を参照できるようにさせる唯一の方法だからである．

図 7.3 は，普通どおり，文の最初の記号の走査をする生成規則から始まっている．つぎに，生成規則 L から M までは，図 6.8 のアルゴリズムの第 2 ステップを実行している．各生成規則は，図 6.7 の 1 行に対応している．もし，生成規則の一つと一致がとれれば，スタック上の変換が，図 6.8 の第 3 ステップで与えたように行われ，プログラムは，再び比較を行うために，生成規則 L に戻る．もし，一致する生成規則がなければ，生成規則 N から N ＋ 2 が図 6.8 の第 4 ～ 6 ステップを実行し，もし，入力されたものが，実際に文であれば，″HALT 0″ と印刷し，文でなければ ″HALT 2″ と印刷して停止する．さもなければ，つぎの記号を走査して，比較を再び行うために L へ戻る．

もちろん，図 7.3 の PL プログラムは，非常に能率がよいというものではない．生成規則の順番を並べ直して，各ステップで行われる比較の回数を減じてやるためにもっと名札を使用してやる．例えば，生成規則 L での変換の後に適用可能な規則は，L ＋ 4 だけである．

ところで，これまで，(1, 1) 限定文脈文法に対する表であればどんな表でも変換できることを説明した．この変換はかなり機械的である．実際，この過程は簡単な修正で (m, n) 限定文脈文法に対する表を PL に翻訳するものに変えることができるだろう．把手となりうる部分記号列の両側に増やしていった文脈にも注意を払わなければならないだろう．これまで述べたさまざまの上向き解析法によって受理される文法はいずれも適当な m と n に対して (m, n) 限定文脈でもあるので，その文法用の上向き認識を行う PL プログラムを書くことが可能である．

```
S:            #        SCAN
L:      a 0 1 #  -->  a W #  GO L
        0 0 1 1  -->  0 W 1  GO L
      0 0 W 1 1  -->  0 W 1  GO L
      a 0 W 1 #  -->  a W #  GO L
        # a W #  -->  # Z #  GO L
          b 0 0  -->  b U 0  GO L
          b 0 1  -->  b U 1  GO L
        b U 0 0  -->  b U 0  GO L
        b U 0 1  -->  b U 1  GO L
        # b U 1  -->  # V 1  GO L
        # V 1 1  -->  # V 1  GO L
        # V 1 #  -->  # V #  GO L
M:        # V #  -->  # Z #  GO L
N:        # Z #              HALT 0
              #              HALT 2
           ANY        SCAN   GO L
```

図7.3 PL で書いた上向き構文解析プログラム

PL による逆戻りのない下向き解析

ここでは，4.3 節の知識を必要とする．CALL と RETURN という2個の〈操作〉によって，生成規則を一まとめにして使用することができるようになる．これは ALGOL や FORTRAN で用いられる手続きに非常によく似ている．これによって，4.3 節にある再帰的下向き技法を用いて，下向き解析法のプログラムを書くことができる．実際，PL 自体，記号処理に都合よく設計されているので，再帰的下向き法をプログラムするのはずっと容易になる．

図7.4 は，文法 (4.3.2) の文を認識するための PL プログラムを示している．この図には，この文法の規則ごとにひと組の生成規則が含まれている．第1行の生成規則は初期設定である．文の最初の記号をスタックにのせて，〈実行文〉に対する生成規則を呼出す．プログラムが，"HALT 2" を印刷して停止したとすれば，入力記号列は，文ではなかったことになる．

生成規則の組は，再帰的サブルーチンと同じような約束に従っている．ひと組の生成規則が呼出されると，検出すべき句の最初の記号は，スタック上にあり，戻る時には，その句に続く記号が，スタック上にある．生成規則の各組を，それに対応する 4.3 節の手続きと比較してみれば，両者は，驚くほど似ていることに気付くだろう．違っているのは表記法だけである．

```
S:      #                                    SCAN CALL STATE
S0:     # 〈実行文〉 #                        HALT 0
        ANY                                  HALT 2

STATE: IF                                    SCAN CALL EXPR GO ST1
        ANY                                  CALL VAR SCAN CALL EXPR
〈変数〉 $:=〈式〉 ANY    → 〈実行文〉 ANY      RETURN
ST1:   IF 〈式〉 THEN    →                    SCAN CALL STATE GO ST2
        ANY                                  HALT 2
ST2:   〈実行文〉 ELSE    →                    SCAN CALL STATE
       〈実行文〉 ANY                          RETURN
        ANY                                  HALT 2

VAR:   I                                     SCAN
       I (                                   SCAN CALL EXPR GO VAR1
       I ANY            → 〈変数〉 ANY        RETURN
VAR1:  I ( 〈式〉 )      → 〈変数〉            SCAN RETURN
       ANY                                   HALT 2

EXPR:  ANY                                   CALL TERM
       〈項〉   +        →                    SCAN GO EXPR
       〈項〉   ANY      → 〈式〉 ANY          RETURN
       ANY                                   HALT 2

TERM:  ANY                                   CALL FACTOR
       〈因子〉   *      →                    SCAN GO TERM
       〈因子〉   ANY    → 〈項〉    ANY       RETURN
       ANY                                   HALT 2

FACTOR: (                                    SCAN CALL EXPR GO FAC1
       ANY                                   CALL VAR
       〈変数〉 ANY      → 〈因子〉   ANY       RETURN
FAC1:  ( 〈式〉 )        → 〈因子〉            SCAN RETURN
       ANY                                   HALT 2
```

図7.4 PL で書いた下向き構文解析プログラム

PL によるプログラム作成

完全な上向き認識法か，完全に下向きの（逆戻りのない）認識法のいずれかを PL でプログラムしているかぎり，どちらも，最も能率のよいプログラムというわけにはいかないだろう．これまでに見たように，上向き認識法が，非能率なのは，各ステップで，あまりにも多くの比較を行わなければならなかったからである（これは，比較の回数を減らすために生成規則を並べ変えて，GO 操作を変更しても，同様であろう）．一方完全な下向き認識法は，多数の

CALL と RETURN を実行するので，単純な GO 操作よりずっと遅くなる．

最もよい解決法は，両者を適当に混ぜることである．原始言語の主要な単位に対してサブルーチン（すなわちサブルーチンとして用いられる生成規則の組）を書く．例えば，ALGOL についていえば，実行文に対するサブルーチン，宣言に対するもの，算術式に対するものなどである（このように分離すると，プログラムを完全に独立な部分にわけて，何人かの人でプログラムできるようになることに注意）．これらのサブルーチンは，何個所かで他のサブルーチンを呼出しているかもしれない．実行文用サブルーチンは，算術式用サブルーチンを呼出すし，実際，部分実行文を構文解析するときには，自分自身を呼出している．

各サブルーチンの中では，上向き法の方が，書き易いことになりがちであることは，図7.1 のとおりで，これはすでに議論した．各ステップで比較される生成規則の数を制限するために，文脈と GO 操作を利用しようというのがねらいである．

全て同じ左辺を持った書換え規則の集合に対応するサブルーチン群の中で，通常，どの選択枝にするかを決定するために最初の記号を検査するいくつかの生成規則から始めて，その選択枝を構文解析するための生成規則の部分集合へ分岐する．〈実行文〉に対するサブルーチンは，この好例である．ある文法の一部

```
〈実行文〉::= IF 〈式〉 THEN 〈実行文〉
            | 〈変数〉:=〈式〉
            | FOR 〈変数〉:=〈繰返し要素の並び〉
              DO 〈実行文〉
            | READ (〈変数〉)
```

を考えよう．これに対する PL プログラムは下記のように書けるだろう．ここで〈変数〉は，重要な構文要素と考えて，サブルーチンで認識していることに注意しよう．このサブルーチンは，非常に多くの場所で用いられているからである．もし，普通の GO VAR を用いると〈変数〉を認識したあと，文脈に応じて，どこに戻るべきかを決めるために，長い系列の生成規則が必要となろう．

```
STATEMENT: IF        SCAN CALL EXPR GO CONDL
           FOR       SCAN CALL VAR GO FORL
           READ      SCAN GO READL
           ANY       CALL VAR GO ASSL
CONDL:  (条件文を認識する生成規則の組)
FORL:   (FOR ループを認識する生成規則の組)
READL:  (READ 文を認識する生成規則の組)
ASSL:   (代入文を認識する生成規則の組)
```

クラスの名を用いることによって，プログラム上の手間と実行時間を大いに節約することができる．下記のプログラムとの差を見てほしい．

```
REAL     →  EXEC 1 SCAN GO IDLIST
INTEGER  →  EXEC 2 SCAN GO IDLIST
BOOLEAN  →  EXEC 3 SCAN GO IDLIST
STRING   →  EXEC 4 SCAN GO IDLIST
COMPLEX  →  EXEC 5 SCAN GO IDLIST
```

に対して

```
CLASSNO  TYPE REAL 1 INTEGER 2
         BOOLEAN 3 STRING 4 COPMPLEX 5;
TYPE  →  EXEC TYPE SCAN GO IDLIST
```

後者の方がずっと書き易いし，読み易い．

演習問題 7.2

1. 図7.3 の PL プログラムを，生成規則を並べ換え，GO 操作を変更して，より能率のよいものにせよ．

2. 下記の文法の文を (1) 下向き法，(2) 上向き法 を用いて認識する PL プログラムを書け．

$$Z ::= b\,M\,b \mid M\,c, \quad M ::= (L \mid a, \quad L ::= M\,a)$$

7.3 意味付けルーチンの呼出し

この節では，EXEC 操作の使用法を説明する．ここで議論することがらは，まだ意味付けルーチンの議論をしていない段階では，あまり明解でないかもし

れない．第12章，第13章を学んでから，この節をもう一度読み直すことを奨めたい．

PL が動いているところでは，実際には，並列に働く少くとも2個のスタックがある．一つは，構文用スタック SYN で，これは，すでに議論した．もう一つは，意味用スタック SEM と称せられ，構文用スタック上の記号の“意味”を保持するものである．すなわち，SEM(i) には，SYN(i) にある記号の“意味”が入っている．この意味とは何かについては，第12章，第13章を読めば明らかになろう．

大部分の終端記号のもとの意味は，定義されていない．すなわち，走査部が記号を SYN にのせるとき，それに対応する SEM の番地には，新しい値は入らない．しかしながら，終端記号 I と N のときはそうではない．走査部が，名前（整数）を走査して，構文用スタックの上に I(N) をのせると，同時に，SEM の上には，その名前（整数）をのせる．(これは名前または整数へのポインタになることもあるかもしれないが，ここでの議論には関係がない.) したがって，I(N) の意味は，I(N) が参照している，名前（整数）である．

PL の動いているところでは，それに関連する意味言語も存在し，この言語で意味付けルーチンがプログラムされる．これは，アセンブリ語，FORTRAN, PL/I, ALGOL かもしれないが，それはどうでもよい．意味付けルーチンとは意味言語で書かれた何らかの方法で識別可能な実行命令の組である．それぞれは，別々の手続であってもよいし，大きな CASE 文の一部であってもよいし，あるいはこの意味言語が，意味付けルーチンの初めを識別するための特別な″意味を持った名札″を与えているといってもよい．

意味言語の中からでも，意味用スタック SEM を使用できる．例えば操作 EXEC 5 が PL 生成規則の中で実行されるときに，意味プログラム中の意味付けルーチン番号 5 が実行される．このルーチンの計算は，SEM の最上段の要素と，意味言語の中で宣言された諸変数を用いて，実行されて，戻ってくる．

例えば，生成規則

```
I → F EXEC 5 SCAN GO LT
```

が只今となり，スタックは

SYN	SEM
I	AB
*	*の意味
T	Tの意味
:	:

のようになっていると仮定しよう．したがって，左辺は，スタックと一致し，SYN の最上段の要素は I から F に変更されて，ルーチン 5 が呼出される．SEM の最上段には，AB があり，これを記号表で引いて，この表での名前 AB のエントリ番号3（とする）を SEM の最上段にのせる．すると

SYN	SEM
F	3
*	*の意味
T	Tの意味
:	:

が得られる．こうして，意味付けルーチン 5 から戻り，つぎの記号を走査して，生成規則 LT が只今になる．

このほかにここで議論すべきことがらは，操 作 EXEC 〈クラスの名〉だけである．クラスの名は，生成規則の左辺にも現れなければならいことを想い起そう．この操作を実行すると，生成規則の左辺にある〈クラスの名〉と一致した，現在スタックの上にある記号を，〈クラスの名〉の宣言によって結び付けた番号の意味付けルーチンが実行される．これは，言葉でいうより，例を示した方がよい．つぎの宣言

```
CLASSNO OPER *1 +2 -3 /4;
```

と，只今の生成規則 E OPER E → E EXEC OPER を考えよう．もしスタックに E − E が入っていれば，ルーチン3が実行されよう．

7.4　参考文献概説

生成言語はもとをただせば，Floyd (61 b) である．彼の論文で導入された言語は，記号処理を目的とし，表記法を別にすれば，PL と大変よく似ている．この言語は Evans(64)によってさらに発展された．PL は FSL——Feldman (66) によって設計作成されたトランスレータ作成システム (TWS)——の中で，構文解析用に使用されている．これはまた，このほかいくつかの TWS でも使用されている．適当な文法が与えられると，その文法の構文認識用の PL プログラムを生成する自動作成機 (constructor) がいくつか設計されている（例えば Earley など）．Ichbiah と Morse (70) には，順位文法から“最適な”生成規則を生成するアルゴリズムが述べられている．彼らの用いた技法では，CALL と RETURN という操作を使用していない．

第 8 章

実行時における記憶域の構成

本章では，よく知られた三つの高水準言語の，実行時における構成を説明する．これによって，プログラム用言語の中にあるいろいろな概念を述べることになろう．まず，FORTRAN のような言語に対する単純な記憶域の構成を示し，つぎに ALGOL のような言語に対する中程度の複雑さをもった構成を示し，最後にPL/Iのような，構造体やポインタ変数，ALLOCATE 文や FREE 文をもった言語に対する，最も複雑な記憶域の構成を示すことにする．

もちろん，ここでは，これらの言語について，すみずみまで述べることはできない．本章の目的は，記憶場所の管理に関する基礎的な事項を詳しく述べ，さらに，一般的な事項をいくつか加えることにある．読者はここで与えた材料をよく理解した上で，どんな言語でもよいから選んで，自分でその言語の実行時の管理ルーチンを設計できるようになってほしい．本章を読んで，何か一つでよいからしっかり身につけてほしいのである．もし，対象となるプログラム言語を十分に理解していないと，注意をひかずに読み過ごしてしまうような事項が数多く含まれている．

言語の中には，動的な記憶割付機構を必要とするものがあり，そこでは，実行につれて一連の記憶場所が，割付けられ，解放され，そして再び別の目的のために割付けられるということが，ランダムに発生する．このような機構を完全に議論するのは本書の範囲外である．完全を期すために，8.10節で，一つの記憶割付法を与えているが，このような技法を学ぶには，Knuth (68) を読むことを奨めたい．まず，つぎの2個のルーチンを仮定しよう：

GETAREA(ADDRESS, SIZE), FREEAREA(ADDRESS, SIZE).

GETAREA は，SIZE 個からなる一連の記憶場所を，その先頭番地 ADDRESS

とともに，呼出したプログラムに割付けるルーチンである．FREEAREA は，ADDRESS 番地から始まる SIZE 個の記憶場所をシステムに返却するルーチンである．システムの中には，記憶場所は割付けられるばかりで，陽に返却することを決してしないものもある．ここでは，これ以上記憶場所の割当てが不可能になると，システムは，**くず拾いルーチン** (garbage collector) を呼出し，それが，実際には，使用されていない場所を探し集めて，自由領域に戻してやるのである．そしてさらに，使用中の場所をつめ合せる——すきまなく一つづきにするために使用中の全てのデータを移動する——こともやらなければならないかもしれない．そうすれば，細分化された，たくさんの使用可能領域のかわりに一つづきの大きな領域が使えるようになるのである．この問題も8.10節で簡単にふれる．このような仕事をするために FREEGARBAGE というルーチンも仮定することにしよう．

本章の構成はつぎのとおりである．8.1節は，データ領域という概念を導入し，全体を通じて，説明の統一を試みる．8.2節は，テンプレートについて述べる．8.3節～8.6節は，計算機の上で，データ構造を実現する方法を易しいものから難かしいものの順に論じている．8.7節は，手続き呼出しと，実引数と仮引数の対応について述べている．8.8節と8.9節では，それぞれ FORTRAN と ALGOL の関連する問題を述べ，8.10節で動的記憶割付とくず拾いを論じてこの章を終る．

8.1　データ領域と見出し表

データ領域 (data area) とは，適当な方法でひとまとめにされたデータのために設けられたひとつづきの記憶場所の集り——主記憶のひとかたまり——である．つねにというわけではないが，あるデータ領域に含まれる全ての記憶場所が，原始プログラムの中で，同じ**有効範囲** (scope) をもっていることが多い．すなわち，これらの記憶場所は，同じ1組の命令系列によって参照することができる（したがって，有効範囲というのは，ブロックであったり，手続きの本体であったりする）．**活性化レコード** (activation record) が Wegner (68) によって同じ概念をあらわすために，用いられているが，データ領域の方が，（英語で）字数が短かいので，こちらを使うことにする．

変数が FORTRAN の COMMON 文で宣言されれば，単一のデータ領域，無名共通ブロックと呼ばれるデータ領域，に記憶場所が割付けられる．ALGOL では，主プログラムで宣言された単純変数の実行時の記憶場所は，主プログラ

ムで必要な，一時的な結果のための記憶場所とともに，一つのデータ領域を形成する．

コンパイル時には，実行時の変数の記憶場所は，（データ領域番号，オフセット）という一対の値で表されている．ここで，データ領域番号とは，データ領域ごとに異なった番号がきめられており，オフセットとは，その領域の先頭番地からの相対番地である．（したがって，データ領域3の先頭番地は，(3, 0)，つぎの場所は (3, 1) のように表す.）コンパイラがある変数を参照する命令をつくりだすときには，この一対の値を，機械の実効番地に変換してやらねばならない．これには，普通，データ領域の**基底番地** (base address，先頭番地の機械上の番地）をレジスタに置き，オフセットの値にレジスタの値を加えて，この変数を参照する．すなわち（領域番号，オフセット）の組は，（基底番地，オフセット）の組に変換されることになる．

データ領域は，静的なものと動的なものの2通りのクラスに分類される．静的なデータ領域の記憶域の大きさは，実行に入る前に事前に決っている．これらの記憶域は，実行を通じて，最初に割当てられたままのものである．したがって，静的なデータ領域にある変数は，（基底番地，オフセット）の組ではなくて，絶対番地で参照できる．

動的なデータ領域は，実行を通じて，つねに実在するわけではない．計算機の中に入ったり出たりし，出てゆくときは，その中の値はすべて失われてしまう．ALGOL の手続きの本体で宣言された変数に対応するデータ領域は，この例である．手続きが呼出されると，GETAREA によって，その手続きに必要な大きさのデータ領域が，主記憶上に割付けられる．手続きの実行を終って，呼出したプログラムに戻る直前に，FREEAREA を呼出して，確保した領域を解放する．手続きに対応するデータ領域は，物理的に同じ記憶場所が，いつも与えられるのではないことを注意しよう．といっても，GETAREA は，つねに，対応するデータ領域の基底番地を返してくるので，手続きは，それをつねに同じ記憶場所（例えば BA）にしまう．すると，データ領域中の任意の変数の番地は，つねに，BA の内容＋オフセットで表すことができる．もし，手続きが再帰的に呼ばれる場合は，呼ばれるたびに，そのときのデータ領域の写しを主記憶の上に残してゆかねばならないので，同じ手続きに対するデータ領域が複数個主記憶上に存在することになる．このようなことが必要なのは，手続きが呼ばれるたびに，それが，新しいデータ領域で仕事をしなければならないからである．このようにデータ領域は，手続きと1対1に対応するのではなくて，

手続きの実行と対応している.

任意の実行時点で，手続きは，複数のデータ領域の中にある変数を参照して構わない．図 8.1 のような構造をもった手続きを含む ALGOL プログラムの

```
BEGIN PROCEDURE A;
      BEGIN PROCEDURE B; BEGIN…END;
            PROCEDURE C; BEGIN…END;
      END;
      PROCEDURE D; BEGIN…END;
   :
END;
```

図 8.1　入れ子になった手続きをもつ ALGOL のプログラム

例を考えてみよう．主プログラムと手続きのそれぞれに別々のデータ領域が対応しているならば，手続き B を実行するときには，B は手続き A と B，および主プログラムに対するそれぞれのデータ領域が参照できる．それぞれに対する参照に何らかの順序をつけるために，参照可能なこれらのデータ領域の先頭番地を集めて見出し表 (DISPLAY) をつくる．いまの例では，手続き B を実行するときに，DISPLAY は

	主プログラムのデータ領域の先頭番地
DISPLAY	手続き A のデータ領域の先頭番地
	手続き B のデータ領域の先頭番地

となっているだろう．したがって，これらのデータ領域を主記憶上につくりだすときに，これらの先頭番地を置いておく，固定領域がいる．するとつぎのような疑問がわいてくる：DISPLAY の表はどこにおけばよいのか．DISPLAY の表にはどんな順序で番地をのせるのか．これらの答は，個々の言語の実際を論ずるところで明らかにしてゆこう．

一般に，プログラムの別々の部分を実行するのに使用するために，実行時を通じて，複数個の DISPLAY が使えるようになっていなければならない．上例で，手続き B が手続き D を呼ぶ場合を考えてみよう．まず D の中で使うために，D と主プログラムのデータ領域の番地を含む DISPLAY を作らなければならない．さらに，すでにある DISPLAY はそのままにしておかなければならない．これは，D の実行を終って B に戻ったときに使うためである．任意の実行

時点をとったとき，その時点で必要なデータ領域を参照するために使われている**活きているDISPLAY**は，たった一つである．

演習問題8.1

1. 図8.1の手続Cを実行するときの活きているDISPLAYはどのようになっているか．手続きAを実行する場合はどうか．図8.5（223ページ）の手続きDの場合はどうか．手続きBではどうか．

8.2 テンプレート

コンパイラが，変数の属性を全て知ることができれば，これらの属性にもとずいて，その変数を参照するために必要な命令を，コンパイル時に，全て生成することができる．しかし，多くの場合，これらの属性は，実行時に動的に決るかもしれない．例えば，ALGOLでは，配列の上限は，コンパイル時には未知でよい．また，言語の中には，実引数と仮引数の型の対応が必ずしも正確でなくてもよいものがある．こんな場合，可能なかぎり属性の対応を許すように目的コードを生成しなければならないので，コンパイラは，単純で能率のよいコードを生成することができない．

この問題を解決するために，コンパイラは，変数だけでなく，その変数の**テンプレート**（template）にも記憶場所を割付ける．そこに実行時にきまる属性が書込まれている．（Websterの学生辞書第7版では，templateを，細工物の形をつくるときの目じるしとして用いられるゲージ，ひな型あるいは型板と定義している．）

このテンプレートは，実行時に属性がきまると書込まれ，変化があれば，書変えられる．

簡単な例を示そう．仮引数が単純変数で，対応する実引数の型が変化してよいものとすると，手続きに渡される実引数は

テンプレート　0＝実数型，1＝整数型，2＝論理型　ほか
値の入っている番地（または値自身）

のようになっていればよいだろう．手続きの中で仮引数を参照するときは，このテンプレートを見て，必要とあれば，型変換を行わなければならない．もちろん，変換のために別のルーチンを呼出してもよい．

多くの場合コンパイラは，このような変数の記憶場所に関する属性が未知なので，その変数の値をしまう場所の割付けを行うことができない．ALGOL の配列がその例である．コンパイラにできるのは，長さのきまったテンプレートをデータ領域の中に割付けることだけである．実行時に記憶場所の属性が既知になると，GETAREA を呼出して，場所を割当て，この番地をテンプレートに書入れる．したがって，データは，つねにこのテンプレートを通じて参照する．構造体やレコードもテンプレートが必要である．実際これらはより複雑で，その構成要素やそのまた部分構成要素の関係を詳細に記述しなければならない．これらは 8.6 節で述べる．

実行時に変更の許される属性がふえればふえるほど，実行時にしなければならない仕事がふえてくる．FORTRAN の目的プログラムが，ALGOL や PL/I のそれに比べて効率がよいのは，全ての属性がコンパイル時にわかるので，実行時にはテンプレートが不要となり，したがって，それを解釈することも要らないからである．

8.3 基本的なデータの型に対する記憶場所

原始プログラム中のデータの型は，計算機の上で同等なデータの型に写さなければならない．整数型や実数型のように，計算機の1語に対応するものもあるし，数語を必要とするものもある．これらをつぎのようにまとめておこう：

1. 整数型変数は，データ領域中の1語，あるいは1記憶単位をとるのが普通である．値は，その機械の標準的な整数の内部表現でたくわえておく．
2. 実数型変数は，1語におさめるのが普通である．もしそれ以上の精度がほしければ，計算機の倍精度浮動小数点数の形式を用いることができる．浮動小数点数の形式を持たない計算機では，2語を用い，1語は指数部，1語は（正規化された）仮数部とすることもある．この場合浮動小数点演算は，サブルーチンで実行しなければならない．
3. 論理型変数は1語におさめることができる．0で偽 (FALSE)，1で真 (TRUE) が表せる．真理値の個々の表現法は，論理演算の機械命令で使われる表現できめられる．論理変数には1ビットを割当て，いくつかの論理型の変数や定数を1語につめ合せてしまうことも可能ではある．
4. ポインタは，他のデータを参照する場合の番地を意味する．ポインタを計算機で実現するためには，ひきつづく2語を使わねばならないこともある．

1語は，指示されているデータのテンプレートを指し，他の1語は，実際のデータを指す．これが必要なのはポインタを使うたびにポインタが指すデータの型が変化してよい場合に備えるためである．

8.4 配列のための記憶場所

配列やベクトルの個々の要素は，主記憶上の1語を占めることにする．2語以上を必要とする場合への一般化は，読者に任せよう．

ベクトル（1次元配列）

ベクトルの要素は，データ領域の中の場所に番地の小さい順あるいは大きい順に割付けるのが普通である．この順序は，計算機とその命令構成の都合による．例えば，IBM 7090 では，基底番地からインデクスレジスタの内容を引いた値が，実効番地となっていたので，配列の要素は，番地の大きい方から割付けられた．ここでは，番地の小さい方から割付ける方が考えやすいので，以後こちらの方を使うことにしよう．例えば A[1 : 10] という配列は，A[1], A[2], ···, A[10] の順に割付ける．

行列（2次元配列）

2次元の配列を割付ける方法は，数通りある．普通は，行方向に割付ける．すなわち，A[1 : M, 1 : N] という配列は，
A[1, 1], A[1, 2], ···, A[1, N], A[2, 1], ···, A[2, N], ···, A[M, 1], ···, A[M, N] となる．A[i, j] 要素は，ADDRESS (A[1, 1])+(i−1)∗N+(j−1) 番地† になることがすぐわかる．これを

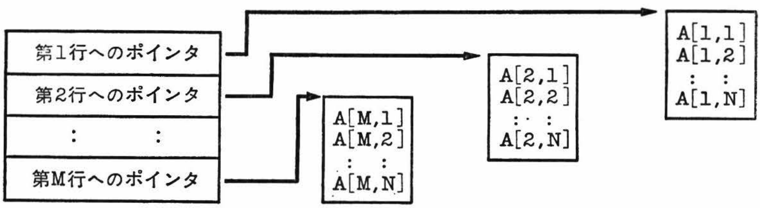
(8.4.1)　(ADDRESS (A[1, 1])−N−1)+(i∗N+j)

と書くことにしよう．第1項は，一定で，1度計算しておけばよい．したがって A[i, j] を見出すには，乗算1回と加算2回を要する．FORTRAN では配列は列方向に割付けなければならない．

第二の方法は，行ごとに別々のデータ領域に割付ける方法で，それぞれのデータ領域へのポインタを要素とするベクトルが必要である．各行の要素は，添字の小さいものから順に連続した場所に割付けられる．したがって，A[1 : M, 1 : N] と宣言すると

---

† 訳注　ADDRESS (a) により変数 a の記憶番地を示す．補遺 (535 ページ) 参照．



が作り出される．各行へのポインタを要素とするベクトルは，その配列を宣言しているブロックまたは手続きに対応するデータ領域の中にたくわえられ，一方，配列自身は，別のデータ領域にたくわえられる．配列要素 A[i, j] の番地は，CONTENTS (ベクトルの先頭番地$+i-1)+(j-1)$ で示される．†

乗算が不要なのがこの方法を使う理由の一つである．全ての行が，同時に主記憶上になくともよいというのも別の理由である．ある行が主記憶上にない場合，ハードウェア上かソフトウェア上の割込がかかるような値をその行へのポインタに持たせておけばよい．割込みがかかると，必要な行を主記憶上に持ってきて，他の行と置換える．この方法はバロース社の B 5500 の拡張 ALGOL で使っている．もし，全ての行が主記憶上にあるときは，ポインタ用ベクトルの分だけ，余分に記憶場所をとっていることになる．

第三の方法は，スパース（粗）行列（要素の大部分が0である行列）に適用される．配列要素 A[i, j] の i と j の値にもとづいたハッシュ技法を用いることができるので，0でない要素だけをテーブルの上にのせることができる．(ハッシュ技法については，第9章を参照)．

多次元配列

ALGOL で

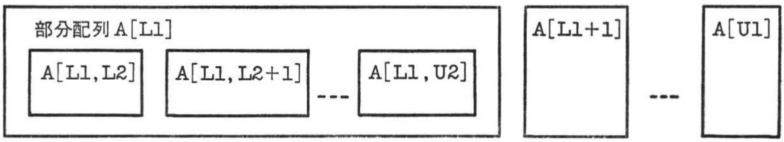
$$\text{ARRAY A}[L1 : U1,\ L2 : U2,\ \cdots,\ L_n : U_n]$$

と宣言された多次元配列の記憶割付法とデータの参照方法を考察しよう．一つの方法は2次元の場合の一般化である．もちろんこれは1次元の場合にも適用できる．

部分配列 A[i, ∗, ∗, ⋯, ∗] は，A[L1, ∗, ⋯, ∗], A[L1+1, ∗, ⋯, ∗], ⋯, A[U1, ∗, ⋯, ∗] の順に並んでいるとする．部分配列 A[i, ∗, ⋯,

† 訳注　CONTENTS (i) により i 番地に入っている値を示す．補遺 (535 ページ) 参照．

$*$] の中に，再び部分配列 A[i, L2, $*$, $\cdots$, $*$], A[i, L2+1, $*$, $\cdots$, $*$], $\cdots$, A[i, U2, $*$, $\cdots$, $*$] がある．これを，各添字ごとに繰返す．このように，添字の位置を進めていくと，最後にある添字がもっとも早く変化する：



A[i, j, k, $\cdots$, p, q] 要素の参照方法はつぎのとおりである：

$$d1 = \mathrm{U1} - \mathrm{L1} + 1,\ d2 = \mathrm{U2} - \mathrm{L2} + 1,\ \cdots,\ d_n = \mathrm{U}_n - \mathrm{L}_n + 1$$

と書くと，d_i は，i 番目の添字のとりうる場合の数を表す．2次元の場合と同様に，部分配列 A[i, $*$, $\cdots$, $*$] の先頭番地は，

$$\mathrm{BASELOC} + (i - \mathrm{L1}) * d2 * d3 * \cdots * d_n$$

で与えられる．ここで BASELOC は，配列の第1要素 A[L1, L2, $\cdots$, L_n] の番地であり，$d2 * d3 * \cdots * d_n$ は，部分配列 A[i, $*$, $\cdots$, $*$] の大きさである．この下の部分配列 A[i, j, $*$, $\cdots$, $*$] の先頭番地は，これに

$$(j - \mathrm{L2}) * d3 * \cdots * d_n$$

を加えればよい．このようにして最後は，次式が得られる．

$$\begin{aligned}(8.4.2)\quad \mathrm{BASELOC} &+ (i - \mathrm{L1}) * d2 * d3 * \cdots * d_n \\ &+ (j - \mathrm{L2}) * d3 * \cdots * d_n \\ &+ (k - \mathrm{L3}) * d4 * \cdots * d_n + \cdots \\ &+ (p - \mathrm{L}_{n-1}) * d_n + (q - \mathrm{L}_n)\end{aligned}$$

共通因子をまとめると

$$(8.4.3)\quad \mathrm{CONSPART} + \mathrm{VARPART}$$

が得られる．ここで

$$\begin{aligned}(8.4.4)\quad \mathrm{CONSPART} &= \mathrm{BASELOC} \\ &- ((\cdots((\mathrm{L1} * d2 + \mathrm{L2}) * d3 + \mathrm{L3}) * d4 + \cdots + \mathrm{L}_{n-1}) * d_n \\ &+ \mathrm{L}_n)\end{aligned}$$

$$(8.4.5)\quad \mathrm{VARPART} = (\cdots((i * d2) + j) * d3 + \cdots + p) * d_n + q$$

である．CONSPART の値は，一度だけ計算してたくわえておけばよい．なんとなれば，これは，配列の上限と下限（その中に配列がおさまる）だけから計算できるからである．VARPART は，添字 $i, j, \cdots, q$ と各次元で添字がとりうる場合の数 $d1, \cdots, d_n$, とから計算する．計算は，大変面倒のように見えるけれど，上のような形の式にしたのは，つぎのような代入文を実行することによって上の式が容易に計算できるからである：

```
VARPART:= 第1添字 i
VARPART:= VARPART * d2 + 第2添字 j
VARPART:= VARPART * d3 + 第3添字 k
              :              :
VARPART:= VARPART * dn + 第n添字 q
```

気が遠くなりそうな式でも，上のような機構を使えば，添字付変数の番地を計算する命令を作り出すのは，きわめて容易なのである．実際どんな次元の配列に対しても，繰返しの性質をこのようにうまく使えば2次元や3次元の配列の番地計算の命令を作り出すのと全く同じたやすさで命令を作り出すことができる．

目的コードの最適化をするような場合には (8.4.2) 式をそのまま使う方がよい．（もちろん一定部分と変化部分は分離しての上であるが．）この式では，d_i が括り出されていないので，計算をいくつかの独立な部分に分けて行うことができるからである．このことについては，第18章で詳しく論じることにする．

ドープ・ベクトル

FORTRAN では，コンパイル時に，配列の上限も下限もわかっている．したがって，上限と下限から $d1, \cdots, d_n$ は定数となるので，コンパイラはこれを用いて，配列の記憶場所の割付けと，配列の要素を参照する命令を生成することができる．ALGOL や PL/I では，上限や下限は，実行時にきめてよいので，これは不可能である．そこで実行時にこれを可能にするために必要な情報を与えるためのテンプレートが必要になる．配列に用いるテンプレートのことを**ドープ・ベクトル** (dope vector) とか，**情報ベクトル** (information vector) とか呼んでいる．ドープ・ベクトルの大きさは，コンパイル時にわかっているので，配列が宣言されているブロックまたは手続きに対応するデータ領域にこのベクトルを割付けることができる．配列自体の記憶場所は，実行時にその配

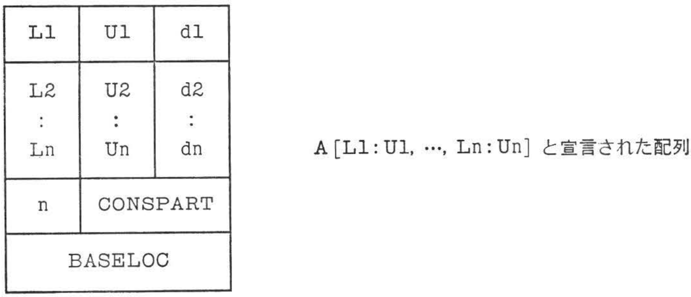
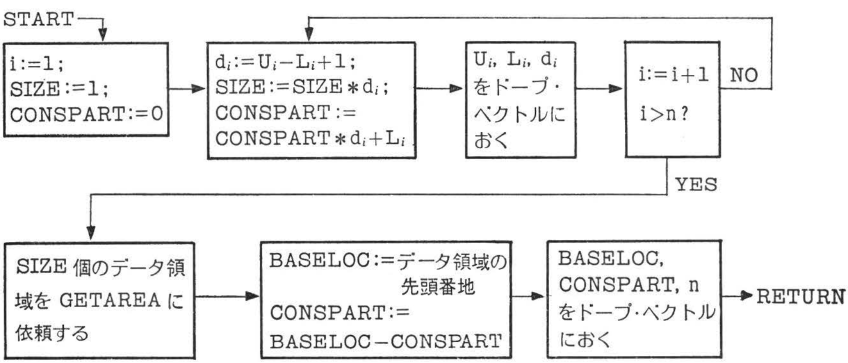


図 8.2 配列用ドープ・ベクトル

列が宣言されているブロックに実行の制御が移るまで割付けることはできない．実行がブロックに入ると，配列の上限下限の値が評価され，配列を割付けるルーチンが呼出される．このルーチンが，必要な記憶場所の大きさを計算し，GETAREA を呼出して，データ領域の割当てを受ける．そして，ドープ・ベクトルの中に必要な情報を全て書込む．配列割付ルーチンの流れを図 8.3 に示す．

ドープ・ベクトルに書込む情報はどんなものであろうか．上にあげた n 次元配列の番地計算機構には，$d2, \cdots, d_n$ と CONSPART の値が必要であった．これが最小限である．もし配列の参照の際に添字の範囲の検査が必要ならば，



このルーチンの引数：n (添字の個数)，下限と上限
L1, U1, …, Ln, Un, ドープ・ベクトルの番地.

図 8.3 配列割付ルーチン

上限と下限の値を憶えておかなければならない．図 8.2 は，必要と思われる情報を示している．

演習問題 8.4

1．（8.4.2）式を，一定部分と添字に依存する変化部分に分離せよ．ただし d_i を括り出してはいけない．この新しい公式を配列要素を参照するのに使うものとすると番地計算を能率よく行うには，ドープ・ベクトルの中にどんな値をいれておいたらよいか．これらの値をドープ・ベクトルにおさめるための配列割付ルーチンの流れ図を作ってみよ．

2．適当な言語で，配列割付ルーチンをプログラムしデバッグせよ．このプログラムを動かすために，手続き GETAREA をダミーの形で書け．

3．配列が列方向にたくわえられているとしよう．配列要素 A[i, j, …, m] の番地計算の式を作れ．配列割付ルーチンとドープ・ベクトルを設計せよ．

8.5　文字列の記憶場所

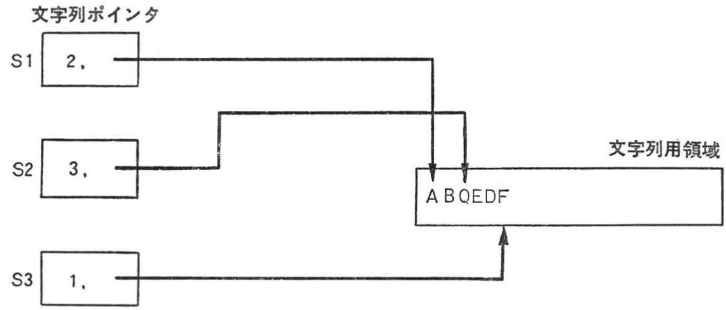
文字列というのは，文字の系列である．もし，系列の長さ（文字の数）が，実行時には不変で，コンパイル時にわかっていれば，その長さに相当する語数の記憶場所を割付けることができる．（例えば，IBM 7090 では 1 語に 6 文字，IBM 360 では，1 語に 4 文字（あるいは 1 バイトに 1 文字）になるように語数をきめる．）ところが，文字列の長さは必ずしも既知ではない．もし，最大の長さがわかっていれば，その長さが入るに十分な語を割付けておき，一つづきの記憶場所の最初の語を，現在の文字数をおさめる語にしておけばよい．

文字列の現在の長さ	
現在の文字列	最大の長さの文字をいれることができる
未使用の場所	

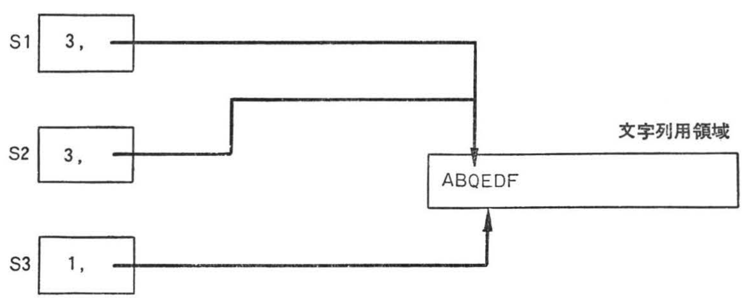
文字列の現在の長さをいれておく語は，テンプレートの一種といってよい．場合によっては，最大の長さを憶えておく語を付加えることも必要になる．

最も融通性があり強力なのは，コンパイル時に，文字列の最大長が不明である場合である．これを計算機で実現するのが最もむずかしい．文字列の代入文 A:＝B の実行では，右辺 B の文字列の長さが，A がすでに占めている文字列

の長さをこえるときに事態が複雑になる．一つの解決法は，IBM 360 に XPL と SPL を実現するときに使われた方法で（McKeeman ほか（70）参照），全ての文字列データのために静的なデータ領域を**文字列用領域**（string space）として大きくとっておき，文字列変数は，文字列の現在の長さと，文字列用領域中の当該の文字列へのポインタとをおさめる2語からなるテンプレートを持っている．このようなテンプレートを**文字列ポインタ**（string pointer）と呼ぶ．文字列代入文 S1 : = 'AB', S2 : = 'QED', S3 : = 'F' を実行すると，文字列ポインタは，つぎのようになる：

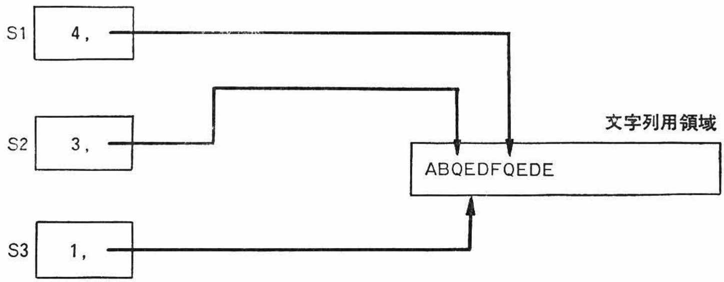


S1 : = S2 のような代入文を実行すると，文字列はそのままで，文字列ポインタだけが変化する．S3 : = SUBSTR (S2, 1, 1) のような代入文も同じように実行できる．例えば，これら二つの代入文を実行すると上の図はつぎのように変化する：



三つの文字列ポインタは全て同じ文字列の一部を指していることに注意．文字列ポインタだけが変化し，文字列自体はそのままなので，単純な代入文は能率よく実行できる．文字列の連接（例えば S1 : = S2 CAT S3）は，自由領域の

中に新しい文字列を作り上げなければならないので遅くなる．この文を実行するとつぎのような図ができあがる．



このようにしていくと，文字列用領域はいつか一ぱいになってしまう．しかしその中の多くは，くず（garbage）——すなわち，もはやどの文字列ポインタからも指示されていない文字列の集り——かもしれない．上の図で，文字 A，B はくずである．領域が一ぱいになると，FREEGARBAGE が呼出されて，くずを見つけ，それをとり除いて，必要な情報だけを文字列用領域の前の方からつめ合せる．

8.6　構造体の記憶場所

すでに定義されたデータの型を集合して新しいデータの型を定義する方策がいろいろと考えられてきた．そのような型を持ったデータを**構造化データ**（structured value）と呼ぶ．COBOL, PL/I, ALGOL W, ALGOL 68, SNOBOL などは，全て，何らかの構造化データを定義する手段を有している．それらの使い勝手はさまざまで，それらを計算機に実現するときの効率も多様である．これらのうちから3種を選んで，それらを実現する方法を簡単に述べてみよう．論点はつぎの4点である：

構造化データに記憶場所を割付ける方法，
構造化データの構成方法，
構造化データの構成要素を参照する方法，
割付けた場所を解放する方法．

Hoare のレコード [Wirth, Hoare (66) 参照]

新しいデータの型は，つぎのように定義する：

RECORD〈名前〉(〈構成要素〉,〈構成要素〉, …,〈構成要素〉)

ここで〈構成要素〉は〈単純な型〉〈名前〉であり〈単純な型〉とは，REAL INTEGER, POINTER などのようにそれぞれの言語で，基本となる型のことである．(ALGOL W では，POINTER のかわりに，REFERENCE を使う.) 例えば

```
RECORD COMPLEX (INTEGER REALPART, INTEGER
IMAGPART)
```

と書いて，COMLEX 型を定義し，これは，REALPART, IMAGPART と名付けられた 2 個の INTEGER 型の要素からなっていることを示している．コンパイル時に，全ての構成要素の属性がわかり，ポインタについても，それが指示できるデータの型がわかっているものとすると，実行時において，構造体自身や，その構成要素に対してテンプレートは不要となり，能率のよい命令を生成できる．n 個の構成要素から成る構造化データは，つぎのような形式で記憶される：

構成要素 1	構成要素 2	・・・	構成要素 n

全ての属性がコンパイル時にわかっており，各構成要素の記憶場所の大きさもわかるので，コンパイラは，各構成要素の相対的な番地をきめることができる．くず拾いの目的からすれば，それぞれのデータ型に対して（プログラマが定義した型を含めて）別々の番号を与えて，各ポインタはテンプレートを持つのが一番よい．このテンプレートには，現在指示されているデータの型を意味する番号がのっている．

ポインタとそのテンプレートの記憶場所は，それに関係するデータ領域の中にコンパイラが割付ける．ともに大きさがきまっているので面倒は起らない．単一の静的なデータ領域を，現在の構造化データを置くために使用することができる．この記憶場所の割付には，特別の GETAREA ルーチンを用いる．ALGOL W は記憶場所を解放するための文を陽に含んでいないので，割当てる場所がなくなると，システムが自動的に FREEGARBAGE を呼出す．不要な場所（くず）を発見できるためには，構造化データの構成要素になっているポ

インタを含めて，全てのポインタがどこにあるか，知っていなければならない．

P:=COMPLEX(S1, S2)

を実行すると，S1 と S2 を構成要素とする COMPLEX 型の新しいデータが作りだされる．ここでPは，COMPLEX 型のデータへのポインタと宣言されているとする．実行時には，この文の実行は，GETAREA を呼出し，必要な記憶場所を得て，S1 と S2 の値をそこにしまう．そして新しいデータの番地がポインタPにのる．

構造化データは，宣言ずみのポインタを通じてのみ参照できる．

P1:=P;P:=COMPLEX(S3, S4)

を実行すると，P1 に P の値が置かれるので，P にあった値は失われない．P1:=P の実行の直後は，P1 と P は同じデータを指していることに注意してほしい．

一方，構成要素の方は，

<構成要素名>(“構造化データへのポインタ”)

によって参照できる．したがって，IMAGPART(P):=IMAGPART(P)+1 とすれば，Pが指示する構造化データの第2成分に1が加えられる．この例を IBM 360 の機械命令で記述してみるとつぎのようになるだろう．

L	1, P	P の値をレジスタ 1 にのせる．
L	2, 4(1)	IMAGPART をレジスタ 2 にのせる．ここでオフセット 4 はコンパイル時に既知．
A	2,=F'1'	レジスタ 2 に 1 を加える．
ST	2, 4(1)	結果を IMAGPART にしまう．

PL/I の構造体

PL/I の構造体は，構成要素がまた構成要素を持つことができるので，Hoare のレコードより複雑である．例えば

```
DECLARE 1 PERSON,
          2 NAME CHARACTER(20),
          2 FATHER POINTER,
          2 MOTHER POINTER,
          2 CHILDREN,
            3 OLDEST1 POINTER,
            3 OLDEST2 POINTER,
            3 OLDEST3 POINTER,
          2 AGE FIXED;
```

構造体は，木構造になっており，節点に名前が付き，端の節点が値を持つ．上例で根にあたる PERSON を水準1にあるという．根から i—1 の長さの道で到達できる節点は水準 i にあるという．

Hoare のレコードと PL/I の構造体の違いが，構成要素が再び構成要素を持つか持たないかという点だけにあるのならば，実行時において両者には本質的な差はないことになる．すなわち，全ての構成要素とその部分構成要素は構造体の先頭番地からのオフセットがコンパイル時にわかるので，これらをあらかじめ割付けてしまうことができる．ところが，PL/I には，さらに二つの拡張がある．

記憶場所を節約するために，ある構成要素に対して CELL 属性を指定することができる．これは，その要素の部分構成要素を全て同じ記憶場所に対応させることを指示するものである.† 任意の時点で値を持つことができる部分構成要素はただ一つであるから，その要素へ値を代入すると，以前に参照されていた他の部分構成要素の値は失われてしまう．例えば

```
DECLARE 1 DATE CELL, 2 A FLOAT, 2 B FIXED;
```

では，要素 A と B が同じ場所を使うことを宣言している．これによってコンパイルが複雑になるが，もし目的プログラムが，それぞれの部分構成要素を参照するたびに，その要素に正しい値が入っていることを確かめなくてよいのならば，実行時の命令の大部分は実際には変えることはない．

もう一つの拡張は，実行時の管理が更に必要なものである．PL/I では，構造体を木とみたてたときに**任意の**節点に当る要素に添字を許している．例えば

† **訳注**　この属性は必ずしも使用できない．記憶場所の対応には DEFINED 属性がある．

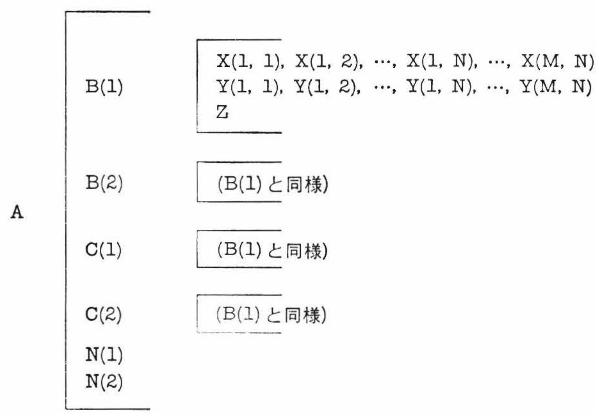
最初に挙げた PL/I の宣言は

```
DECLARE 1 PERSON,
          2 NAME CHARACTER(20)
          2 FATHER POINTER,
          2 MOTHER POINTER,
          2 CHILDREN(3) POINTER,
          2 AGE FIXED;
```

としても，本質的には同じ効果をもたらす．構成要素 CHILDREN は，3個のポインタからなっていて，それぞれ別々に参照できる．このほか

```
DECLARE 1 A, 2 B(2), 3 X(M, N), 3 Y(M, N), 3 Z,
             2 C LIKE B,
             2 N(2);
```

のような構造体の宣言を考えてみよう．これは，つぎのような構造体を示している：



添字の幅をきめる式の値は，実行時に評価されるので，配列の場合と同じように，テンプレートやドープ・ベクトルをそのために用いなければならない．この宣言では，A 自身と Z を除いた全ての要素にたいしてドープ・ベクトルが必要である．

PL/I には，STATIC, AUTOMATIC, CONTROLLED という三つの異なった記憶割付法がある．構造化データとそれ以外のデータとで，記憶割付の指

定の間に差はない．これは ALGOL W でも同様である．PL/I では，CONTROLLED 属性を与えられたデータを自由領域に返却する仕事はプログラマに課せられていることを言っておかなければならない．

特定の構成要素は，その節点に届くまでの節点の名前をピリオドをはさみながら連ねて参照する．したがって，本節の最初の宣言の例で，OLDEST1 を参照するには，PERSON. CHILDREN. OLDEST1 とする．

構成要素に寸法が指定してあると，それに添字をつけなければならない．最後の例で，C (1) の構成要素 X (1, 2) を参照するには，A. C (1). X (1, 2) と書く．これ以外の方法も可能である．構造体の定義に現れる次元が多くなればなるほど，その構造体の型のデータを生成したり，参照したり，削除したりするために，コンパイル時にしなければならない仕事がますますふえてくる．

Standish のデータ構造 [Standish (68) 参照]

単純なものから複雑なものへ，すなわち，計算機で効率よく実現できるデータ構造から，効率は悪いが，豊富で強力なデータ構造へと進むことにしよう．Standish は彼の学位論文で，構造自身が実行時に変りうるような構造を許す記法を ALGOL 60 に埋込むことを与えている．構成要素の寸法ばかりでなく，要素の個数や型さえ動的に変更できるのである．一般に，コンパイル時にわかるものは何もなく，実行時にテンプレートを通じて全てを取仕切らなければならない．そしてこのテンプレートも実行時に作らなければならないのである．例えば，つぎのようなテンプレート手続き

```
TEMPLATE PROCEDURE SEQ (n, t);
SEQ: = n * [t];
```

の実行によって，n 個の型 t を要素に持つ構造体を記述するテンプレートを作りだす．(Standish はテンプレートの代りに DESCRIPTOR を用いている．)

```
D1 :: = SEQ (2, REAL); D2 :: = SEQ (k, D1)
```

を実行すると，二つの実数型の要素を持った構造化データのテンプレートをテンプレート変数 D1 に与え，それぞれ D1 型を持った k 個の要素からなる構造化データのテンプレートを D2 に与える．実行時に，構造化データに記憶場所を割付け，そこに値を代入する方法はいくつかあるが，このように，テンプレートを作りあげる方法も他にいくつかある．これについては，深入りしないが，

実行時に，構造化データ全てに対するテンプレートを，保持しておかなければならない点だけを強調しておこう．事実，このテンプレートは，例えば PL/I の構造体のようなデータのコンパイルを容易にするために，コンパイラが使用している記号表の要素を集めたようなものである．第10章で述べるが，構造を持ったデータは，少くともつぎのような事項を明記した節点を有する木構造として，計算機に実現するのが，最良である．すなわち

1. その節点が端点であるかないか．
2. 端点であれば，その型は何か．
3. 端点であれば，値が定っているときのそれへのポインタ．
4. 端点でなければ，その部分構成要素にあたる節点へのポインタ．
5. その次元と寸法．

あるデータの構成要素を参照しようとするときはいつもテンプレートの解読をしなければならない．根から始めて参照すべき節点への道を見出し，その節点の型を調べ，最後にその値を取出したり，値を代入したりする．

8.7　実引数と仮引数の対応

実引数と仮引数の対応には多くの種類がある．ここでは，それらを述べ，計算機上でそれらを実現する方法を示す．仮引数というのは，手続きの中にあって，手続きが呼ばれると，他の名前あるいは式と置換えられる名前を意味する．FORTRAN で

```
SUBROUTINE A(X, Y)
```

とすると，X と Y が仮引数である．ALGOL では，同じものを PROCEDURE A(X, Y) とする．手続きを，例えば A(B, C*D) と呼出すと，仮引数 X, Y は，それぞれ**実引数** B, C*D と何らかの方法で結びつけられる．

どんな言語であれ，手続きが呼出されると，実引数の番地の一覧表が手続きに引渡される．手続きはこれらの番地を自分自身のデータ領域に写しとって，実引数と仮引数の対応を付けるために用いる．実引数のほかに，プログラマが陽に気づいていないいくつかのかげのパラメタがあるのが普通である．その一つはもちろん戻り先番地である．（全てのインデクス・レジスタの値をこの一覧表の中に退避しておかなければならないことがあるかもしれない．しかし，これは機械ごとに異なるので，このような処理については議論しない.）いろ

いろの言語の中で，いろいろな種類のパラメタを調べると，上記以外に言語に依存するパラメタとしてどんなものが必要かについて後で見てみたい．

結局，呼ばれた手続きに渡される一覧表はつぎのようなものである：

かげのパラメタ　1
　　　　:
かげのパラメタ　m
実引数の番地　　1
　　　　:
実引数の番地　　n

全てのパラメタが，実際に一覧表に入って渡されるとはかぎらないことに注意．便宜上上のように示している．例えば，戻り先番地は適当なレジスタに入って渡されるのが普通である．

一覧表の中の番地は，何を表しているのだろうか．それは，言語と引数の種類に依存する．これをつぎに議論する．ここで議論する引数の種類を以下に挙げる：

1. 番地呼び (call by reference)
2. 値　呼　び†(call by value)
3. 結果呼び (call by result)
4. 見代り引数 (dummy argument)
5. 名前呼び†(call by name)
6. 実引数としての配列名
7. 実引数としての手続き名

演習問題2と3は，読者の納得がゆくように初めの5種類が，プログラム中の適当な前後関係の中で，全部異なった結果になるように工夫したものである．

番地呼び

この型の結合は，計算機で実現するのが最も容易である．実行時には，手続きの呼出しに先立って，実引数の処理を行う．すなわち，実引数が変数でも定数でもなければ，その式の値を計算して一時的な場所にたくわえる．変数また

† 訳注　ALGOL (JIS) では，これらをそれぞれ値とり，名前変えとしている．ここで述べる引数の種類は，ALGOL だけを対象としているわけではないので，他の引数との関連からこのように訳した．

は定数の記憶場所あるいはこの一時的な場所の番地を求めて，これを呼ばれた手続きに渡す．呼ばれた手続きは，この情報を用いて，値の入っている場所を参照する．例えば

(8.7.1)　PROCEDURE P(X); BEGIN ··· X:＝X＋5; ··· END

という手続をP(A[I])と呼出すことを考えてみよう．ここでAは配列である．手続きを呼出す前にA[I]の番地を計算し，これを実引数の一覧表の中にいれ，手続き側に渡す．X:＝X＋5 の実行により，A[I] に5が加えられる．この文の機械命令はつぎのようなものになるだろう：

1. レジスタ1に実引数の番地をのせる．
2. レジスタ2に 0(1) 番地の値をのせる．
3. レジスタ2に5を加える．
4. レジスタ2の内容を 0(1) 番地にしまう．

ここで 0(1) というのは，[0＋レジスタ1の内容] 番地を意味する．

値　呼　び

この型の結合では，**呼ばれた**手続き側のデータ領域の中に仮引数に対応するデータの場所を持っている．番地呼びの場合と同様に，手続きの呼出しに先立って，実引数の番地が計算され，一覧表におさめられて手続きに渡される．しかしながら，手続きの実行を始める前に，手続きは，その番地から値を取出し，自分のデータ領域の中の場所におさめる．この場所が，手続きの実行中つねに，局所的な変数と同じように用いられる．したがって，手続きの実行の結果，実引数の値が変ってしまうようなことは絶対にありえない．

別の見方をすれば，値呼びの仮引数は，手続きの実行のたびに，実引数の値を初期値とするような，手続きに局所的な変数である．

手続き (8.7.1) の仮引数Xが値呼びであるとすると，この目的プログラムは，つぎのようなものになろう．

```
                  ┌···
                  │ レジスタ1に実引数の番地をのせる．
手続きの初期設定：│ レジスタ2に 0(1) 番地の値をのせる．
                  │ レジスタ2の値をこの手続き内のXの場所にしまう．
                  └···
```

```
            ┌…
手続きの本体:│Xをレジスタiにのせる．5を加える．Xにしまう．
            └…
```

結果呼び

ALGOL W (Bauer ほか (68) 参照) では，RESULT と指定した仮引数Xについて，次のようなとりきめがある：

1. 手続きのデータ領域中に仮引数Xの場所を確保する．手続きは，この場所を変数Xに対する局所的な場所として用いる．
2. 値呼びの引数の場合と同様に，手続きの呼出しの際に，実引数の番地を計算し，それを手続きに引渡す．
3. 手続きの実行が終ると，Xの最後の値を，2.で計算した番地に移す．

いいかえれば，結果呼びの引数は手続きに局所的な変数であって，手続きを抜けだすときこの変数の値が，対応する実引数に引渡される．(したがって，この実引数は変数でなければならない.) 結果呼びという考え方は，ALGOL の名前呼び (後述) を補うために設けられた．すなわち名前呼びによる結合は非常に能率が悪く，プログラムではこれほどの能力を必要としない場合が多いからである．

ALGOL W では同一の引数に対して値呼びと結果呼びの両方を指定することができる．このとき，手続きに局所的な変数Xに，手続きの実行に先立って，初期値として実引数の値が与えられ，Xの最終の値が実引数に返される．FORTRAN II† では，全ての引数は，番地呼び (番地による結合) であった．しかるに FORTRAN IV† では，普通の引数は，値呼びかつ結果呼びとなっている．しかし，プログラマの指定によってそれを番地呼びにすることもできるようになっている．

---

† **訳注** ここで著者が FORTRAN II, FORTRAN IV と称しているのは，IBM 社の FORTRAN を念頭においているものと思われる．FORTRAN の標準規格 (JIS, ANSI など) では，どちらの結合方法をとっても FORTRAN で書いたプログラムが正しく解釈されるような規程を与えている．逆にいえば FORTRAN でプログラムを書く人は，コンパイラがどちらの結合方法を用いていようとも同じ結果をもたらすようにプログラムを書くべきであってコンパイル技法の違いを利用するようなプログラムは書かないようにすべきであろう．これについては，西村編著 JIS FORTRAN 全釈 [オーム社，昭和49年] pp. 168～169 に詳しい．なお現在わが国で用いられている FORTRAN コンパイラでも例えば FACOM 230-45 S 用の FORTRAN S の引数の結合は番地呼びであり，HITAC 8000 シリーズの FORTRAN は値呼びかつ結果呼びとなっているなどさまざまである．

身代り引数

FORTRAN II† では，(8.7.1) の手続き P を，P(3) として，呼出すと，発見が困難な誤りを招くことになる．FORTRAN では，定数3をおさめている場所の番地を手続きに引渡すので，手続き中で，X = X + 5 を実行すると，(番地呼びでは) 3の記憶場所の内容が8になってしまう．手続きから戻って，3を参照すると，実は8の値が入っていることになる．PL/I では，この問題の解決策として，実引数が

1. 定数
2. 式 (変数を除く)
3. 対応する仮引数と異なった属性を持つ変数

の場合に，特別に取扱う．すなわち，呼出し側で，一時的な場所を用意し，実引数を評価して，それを一時的な記憶場所におさめ，この番地を引数の一覧表にのせて引渡す．こうすれば，P(3)によって，定数3は，決して変更されない．

このような，一時的な変数を**身代り引数** (dummy argument) と呼んでいる．一時的な場所を，呼出し側が用意することを除けば，PL/I の身代り引数は，ALGOL の値呼びによる結合と同じような働きをする．

プログラマから見て，さらにもう一つ違いがある．ALGOL では，手続き側で仮引数を VALUE と指定するのに，PL/I では，呼出し側で実引数を値呼びとして用いるか，FORTRAN のような番地呼びとして用いるかを選択することができる．例えば，P(J) と書けば，実引数 J は番地呼びであるし，P((J)) と書けば，実引数Jは値呼びあるいは身代り引数による結合となる．

名前呼び

ALGOL の報告書 [訳注：Naur (63a)] によれば，引数が名前呼びということは，仮引数の現れるところに，実引数を文字通りに，代入することを意味している．すなわち，もし

```
PROCEDURE R(X, I);BEGIN I:= 2;X:= 5;I:= 3;X:= 1 END
```

という手続きを

(8.7.2)　R(B[J * 2], J)

† 訳注　引数の結合を番地呼びで行っている FORTRAN の意．

と呼出すと，R の本体が，

BEGIN J：＝2;B[J＊2]：＝5;J：＝3;B[J＊2]：＝1 END

のように書換えられてから実行を開始する．もちろん，このようにに文字をすっかり代入することを計算機で能率的に実現するのはむずかしいので，これと同等の効果を得る方法を工夫しなければならない．X に対応する実引数は，J の値が変化するたびに変化することに注意しよう．つまり，実引数の番地を，手続き呼出しに先立ってきめてしまうことができない．そこで，手続き中で，仮引数を参照するたびに，実引数の番地を計算し直さなければならない．

名前呼びの引数機構を実現するには，引数ごとに，別々のルーチンを目的コードとして用意するのが普通の方法である．このルーチンは，Ingerman (61a) によって**サンク**(thunk，**訳注**，革ひもの意，thong の訛)と名付けられ定着した．手続きの呼出しに際して，サンク・ルーチンは，実引数が変数でなければ，その引数を評価し，その値を置いた番地を返す．手続きの本体で，仮引数の参照が必要になるたびに，このルーチンが呼出され，ルーチンから返される番地を用いて，参照が行われる．

(8.7.2) の手続き呼出しの目的コードを以下に示そう：サンク・ルーチンが正しく働くためには，おもてむきの引数以外に，かげのパラメタがいくつか必要であることを注意しておこう．これについては，8.9 節に述べる．

GO TO L

A1：［B[J＊2]の番地を求める
サンク・ルーチン］

A2：［J の番地を求める
サンク・ルーチン］

L における手続き呼出しの引数一覧表

L：右の引数一覧表を → 持ってRを呼出す．

［RET (戻り先番地)
A1 (第1実引数用サンクの先頭番地)
A2 (第2実引数用サンクの先頭番地)］

RET：

番地呼びと名前呼びの違いは，つぎのとおりである：番地呼びの実引数は，手続きを呼出す前に，その番地を一度評価するだけである．それに反して，名

前呼びでは，手続きの本体中で，仮引数が参照されるたびに，実引数の番地を評価する．

実引数としての配列名

この場合，対応する仮引数も配列でなければ意味がない．手続き呼出しが行われると，ドープ・ベクトルの番地，または，（ドープ・ベクトルを必要としない言語では）配列の第1要素の番地が，手続きに渡され，手続きはこれを用いて，配列の要素を参照する．

実引数としての手続き名

引渡される番地は，実引数として現れた手続きの最初の実行命令の番地である．もしこの名前自身が，仮引数であれば，この手続きが呼ばれたときに，それに対応する実引数の番地がすでに渡されているはずである．

FORTRAN の手続き呼出しに必要な道具立てはこれだけである．ALGOL や PL/I では，さらに必要なものがある．この議論は，8.9節 ALGOL の実行時の記憶域の管理で行う．

演習問題 8.7

1. 適当な高水準言語を用いて，手続き EXCHANGE(X, Y) を書け．ただし，この手続きは，X と Y に対する実引数（これらはともに変数でなければならない）の値を交換するものである．X と Y とはどんな種類の引数でなければならないか．このような手続きを ALGOL で書くことができるか．**ヒント**：つぎの二つの手続き呼出しを比較してみよ．

```
EXCHANGE(A[I], I), EXCHANGE(I, A[I])
```

2. つぎの ALGOL 型プログラムについて答えよ：

```
BEGIN INTEGER I; INTEGER ARRAY B[1: 2];
      PROCEDURE Q(X); INTEGER X;
      BEGIN I:=1; X:=X+2; B[I]:=10;
            I:=2; X:=X+2
      END;
  B[1]:=1; B[2]:=1; I:=1; Q(B[I])
END
```

つぎの5通りの引数結合について，それぞれこのプログラムを実行したと

き， B[1], B[2] にはどんな値が得られているか：

(1) 番地呼び，(2) 値呼び，(3) 値呼びかつ結果呼び，(4) 結果呼び，(5) 名前呼び．**ヒント：** 1通りについては，結果は不定になる．残りの4通りの場合は，それぞれ異なった結果を得る．

3. 上問のプログラムで X が PL/I 型の引数と仮定して，このプログラムを実行せよ．つぎに，手続き呼出し Q(B[I]) を Q((B[I])) に変えて実行してみよ．I, B[1], B[2] に得られた値は，2通りの実行で異なってくるか．

8.8 FORTRAN における記憶域の管理

FORTRAN では，サブルーチンの中でサブルーチンを定義（サブルーチンの入れ子）したり，サブルーチンを再帰的に呼出したり，動的な配列を用いたりすることを許していない．サブルーチンが入れ子になっていないので，サブルーチンが参照できる変数は，サブルーチンに局所的な（そのデータ領域の中にある）変数と COMMON ブロックにある変数だけである．COMMON 領域は，静的であって，実行に入る前に，固定した領域を割付けることができる．したがって，COMMON 領域中の変数の参照は，原理的には，絶対番地の参照といってよい．

配列の動的割付が許されていないので（DIMENSION 文で与える配列の寸法は，全て定数である†）実行時の記憶場所の管理は不要である．配列はコンパイル時に長さのきまったデータ領域に割付けることができる．

このことは，現実には，見出し表（DISPLAY）が不要であることを意味する．DISPLAY というのは，プログラムの任意の時点で，その時点で参照できるデータ領域の手掛りを得るために必要であることを思いだしていただきたい．ところが，FORTRAN におけるデータの参照は，全てコンパイル時に既知となった番地の参照であることが，わかったのである．

図 8.4 は，FORTRAN の静的なデータ領域の典型的な例である．かげのパラメタは，戻り先番地や，ハードウェアに依存するレジスタ類などで，サブルーチンの呼出しはつぎのような手順を踏む：

1. 各実引数の番地を評価し，引数の一覧表にしまう．（この表は呼出し側

† **訳注** 仮引数が DIMENSION 文の中に現れる場合に整合寸法が許されているが，これによって，動的な割付が行われるわけではない．

のデータ領域の中にある.)

2. 引数の一覧表の先頭番地をあらかじめ約束した大域的な場所（例えば，あらかじめ適当なレジスタをきめておく）に置く.
3. 戻り先番地を一覧表に置いて，手続きに飛ぶ.

サブルーチンの本体を実行する前に，サブルーチンはつぎの手順を踏まなければならない：

1. かげのパラメタを，自分のデータ領域に移す.
2. 実引数の番地を自分のデータ領域に移す.

手続きの実行を終えて戻る際には，退避しておいたかげのパラメタをもとに戻して，戻り先番地に飛ぶ.

かげのパラメタ用の場所
実引数の場所
単純変数，配列， 中間結果用変数

図8.4　FORTRAN サブルーチンの典型的なデータ領域

8.9 ALGOL における記憶域の管理

ALGOL はブロック構造を持っており，手続きの中で手続きが宣言できる構造になっているので，実行時に，適当な記憶域の管理が必要となる．しかし，これは比較的簡単な機構で実現できる．ALGOL はポインタ変数を許していない．これを許すことが，記憶管理の複雑さを増大させる主たる要因の一つなのである．さらにブロックや手続きが入れ子構造を作っているので，あのなじみ深いスタックという，単純な機構を使って，記憶域を管理することが可能となる.

要するに，番地が一つながりとなった大きな記憶域をスタックとして用い，実行のはじめこのスタックは，空である．一つ一つの手続きの実行に応じて，長さのきまった局所変数やドープ・ベクトルなどを，単一の動的なデータ領域にのせてゆく．（ここで主プログラムは，システムから呼ばれる引数を持たない手続きとみなしている.）ある手続きが呼ばれると，その手続きのデータ領域として十分な記憶場所をスタックの先頭番地の上にのせる．手続きから戻るときに，スタックから取除いて (pop)，今使っていた領域を解放する．ブロッ

```
      BEGIN…
        PROCEDURE A(X, Y); PROCEDURE X; INTEGER Y;
            BEGIN    PROCEDURE B(Z); PROCEDURE Z;
                       BEGIN ARRAY F[1:10];
L4:                            Z(F[1]+Y);
                       END;
L3:                  B(X);
            END;
        PROCEDURE C;
            BEGIN    ARRAY G[1:10];
                     PROCEDURE D(W); INTEGER W;
                       BEGIN
L5:                        G[1]:=W;
                       END;
L2:           A(D, 1);
            END;
L1:  C;
      END;
```

図 8.5 入れ子になった手続と並列な手続き

クは，ブロックの中や手続きの中で入れ子になっているので，手続きと同様にする．ブロックに入ると，スタックの最上段に，そのブロックに関係する配列の要素を含んだデータ領域用の場所をのせる．ブロックから出るときは，使用していた場所をスタックからはずす．この機構は非常に簡単なので，GETAREA や FREEAREA といったルーチンは，実際に必要でない．これらのルーチンを呼出さなくとも，記憶の割付や解放は，数ステップの命令でできるからである．

ここで，手続きの結合と，大域的な変数の参照が正しく行われることを確かめておかなければならない．これが本節の議論の主題である．ここに述べる特別な方法は，おそらく，他の文献にあるいろいろな方法より複雑だろう．それは，ブロックへの出入りをできるだけ能率よく行うように努めたからである．この節の演習問題では，別の方法にも言及している．

ここで用いるスタックの概念を説明するために，図8.5に示す構造を持ったプログラムを考えよう．まず，主プログラムのデータ領域に記憶場所を割付ける．名札 L1 のところで手続きCを呼出すと，C のデータ領域が割当てられ，つぎに配列Gのデータ領域が割当てられる．C の本体の実行に入り，名札 L2

で，手続きAを呼出すと，Aのデータ領域が割付けられる．したがって，名札L3の点では，スタックはつぎのような構成になっている：

手続きAのデータ領域	スタックの先頭
配列Gのデータ領域	
手続きCのデータ領域	
主プログラムのデータ領域	スタックの底

以下，本節では，つぎの諸点を順に述べてゆこう：

(8.9.1)
1. 活きている見出し表（DISPLAY）とデータ領域の関係．
2. 手続きのデータ領域の形式．
3. ブロックの開設，配列の宣言，ブロックの閉鎖．
4. 仮引数ではない手続きの呼出し
5. 手続きの初期設定と手続きからの脱出処理．
6. サンク・ルーチンとその呼出し法．
7. 仮引数である手続きの呼出し．
8. 手続き名を実引数として用いた場合の処理．
9. GO TO 文の処理．

活きている見出し表とデータ領域の関係

ALGOLを計算機で実現するには，手続きの実行のたびにそれに対応する動的なデータ領域の処理が必要である．（ここで主プログラムも手続きと考える．）任意の時点で，いくつかの手続きが，実行の状態にある可能性があるが，実際に**活きている**のは，ただ一つであり，ほかはそれが実行を終って戻ってくるのを待っている．活きている手続きに対応するデータ領域と見出し表（DISPLAY）も，活きていると呼ぶことにしよう．この活きている手続きが，入れ子の水準 i にあれば，この手続きは，その手続きに局所的な変数以外に，水準 $1, 2, \cdots, i-1$ にある手続きと，主プログラムにある変数を参照できなければならない．したがって，活きているDISPLAYは，図8.6に与えるような

形式をとることになる.

主プログラム（水準0）のデータ領域の先頭番地
水準1の手続きのデータ領域の先頭番地
:
水準 i の手続きのデータ領域の先頭番地

図 8.6 水準 i にある手続きの活きている見出し表

では，DISPLAY 自体はどこに割付ければよいのだろう．DISPLAY 用に別のスタックを設けるのがよく行われている方法の一つであるが，ここでは，別の方法を採用しよう．それぞれのデータ領域に関連して，DISPLAY が一つあるのだから，そのDISPLAYをそのデータ領域の初めの数語に割当てる．こうすると活きている DISPLAY の先頭番地は，つねに活きているデータ領域の先頭番地と一致する．さらに，ACTIVEAREA と名付けた大域的な場所（例えば，インデクスレジスタを1個それに割当てる）には，つねに，活きている DISPLAY（すなわち，活きているデータ領域）の先頭番地が入っているようにしておく．これを，現在実行中の手続きが使用する．

活きている手続き内からデータを参照するには，つねにACTIVEAREA を通じてそのDISPLAYを用いて行う．

手続きのデータ領域

手続きは，それぞれ，コンパイル時に長さの決るデータ領域を持っている．下にそれぞれの領域に現れるデータの一覧表を出現する順に示す．この一覧表の項目のうちには，ここでは意味のないものもあるが，適当なところで後述する．

1. その手続きの DISPLAY.
2. STACKTOP と名付けられた場所（1語）．STACKTOP には，この手続きのデータ領域が割当てられた直後のスタックの先頭番地が入っている．したがって，この番地は，この手続きのデータ領域の最上段の場所の番地でもある．
3. 4個のかげのパラメタ：

   a） 戻り先番地.

b）　実引数 DISPLAY の番地.[†]

c）　大域的 DISPLAY の番地.[†]

d）　この手続きを呼んだ時点におけるスタックの先頭番地.

4.　実引数の値または番地.

5.　手続き中のブロックごとに：

a）　ブロックの中でスタックの先頭番地を保持する STACKTOP の場所.

b）　単純変数の記憶場所.

c）　ブロック中で宣言されている配列のドープ・ベクトルの記憶場所.

d）　ブロック中で必要な中間結果用の記憶場所.

ブロックの中に現れる単純変数やドープ・ベクトルや中間結果用の記憶場所の並び方は重要ではないので宣言された順序に並べるのが最もたやすい．並行ブロック中の変数は，同時に参照されることは決してありえないので，並行なブロックには同じ記憶場所が使用できる．しかし必ずこうしなければならないわけではない．図 8.7 の手続きを考えてみよう．この手続きに含まれるブロックには，図の右のような番号を付けておく．このデータ領域は図 8.7 の上に示すようになる．ここでブロック 2 の変数は，ブロック 3 およびブロック 4 の変数と同じ記憶場所をわけあっている．

変数の番地付け機構

水準 i にある手続きの中で，変数か仮引数を参照するとしよう．もしこの参照が，この手続きに局所的な変数か，この手続きで宣言した仮引数であれば，その実行時の番地は，

(データ領域中の相対番地) ＋ ACTIVEAREA

となる．もし，この変数や仮引数が，この手続きを囲む水準 k で宣言されているものであれば，その番地は

(データ領域中の相対番地) ＋ CONTENTS (ACTIVEAREA ＋ k)

となる．われわれは，この水準番号 k を，しばしばデータ領域番号の代りに用いることがある．すると同じ水準にある手続きには同じ番号が与えられるが，コンパイラがそれらを同時に処理することはあり得ないであろう．

---

† 訳注　231，234 ページで詳述.

DISPLAY（主プログラムおよび手続き A のデータ領域の番地）	
手続き A の STACKTOP の場所（A のデータ領域の最上段の番地）	
かげのパラメタ	
実引数 X および Y	
ブロック1 の STACKTOP，変数Z，配列 B のドープ・ベクトル	
ブロック2 の STACKTOP，変数D およびE	ブロック3 の STACKTOP，配列A のドープ・ベクトル
	ブロック4 の STACKTOP，変数E

```
PROCEDURE A(X, Y); INTEGER X, Y;
L1:  BEGIN  REAL Z;  ARRAY B[X:Y];
        L2: BEGIN REAL D, E;
        L3:   :
              END;
        L4: BEGIN ARRAY A[1:X];
              L5:  BEGIN REAL E;
              L6:    :
                     END;
        L7: END;
L8:  END;
```

図 8.7　入れ子になったブロック

ブロックの開設，配列の宣言，ブロックの閉鎖

ブロックは，それぞれのブロックの中で使用する単純変数や，ドープ・ベクトルや，中間結果のために一連の記憶場所をまとめてブロックごとに持っている．配列のための記憶場所はブロックの入口で確保され，ブロックの出口で解放される．ブロックの入口と出口での仕事を最小にするためにこれから述べる特別な機構が考えられた．これを使えば，実際，ブロックの開設ではわずか 2 命令ですみ，また閉鎖では 1 命令もいらないのである．

すでに述べたように，配列用の記憶場所は，スタックの先頭に置かれる．そこで，つぎのような規則を作る：すなわち，ブロックが実行される直前に，そ

のブロックの STACKTOP の内容は，このブロックのスタックの先頭の番地となっている．これはつぎのことを意味する：

(8.9.2)　実行時のスタックの先頭は，活きているブロック――現在実行中のブロック――の STACKTOP の値によってつねに定まる．

手続きのデータ領域が割当てられると，そこには，スタックの先頭を指す STACKTOP のための場所があることを思い出そう．これは，手続きの本体を囲む仮想的なブロックにとっても，STACKTOP の場所であると考えることもできる．

ブロックに入ると，規則 (8.9.2) に従うために，その一つ外側のブロック（もし，手続きの主ブロックに入る場合には，それを囲む仮想的なブロック）の STACKTOP の内容を，現在のブロック用の STACKTOP の中に写しとらなければならない．したがって，ブロックの入口の処理は非常に能率がよい．

次のステップは，ブロックの中で宣言されている配列に場所を割当てるものである．配列ごとに，つぎのような処理を行う：

1. 下限と上限を評価する．
2. つぎの3種のパラメタを持って，配列割付ルーチンを呼出す．
   a) 下限と上限の値，
   b) ドープ・ベクトルの番地，
   c) このブロックの STACKTOP の番地．

配列割付ルーチンは，ドープ・ベクトルにおさめる全ての情報を計算し，初期設定する．つぎに，現在のスタックの先頭を指している3番目のパラメタの情報によって，正確な数の記憶場所をスタックにのせる．最後に，この第三のパラメタにスタックの新しい先頭番地を代入する．

ブロックごとに，STACKTOP 変数が別々に必要となる理由をたずねられるかもしれない．それは，ブロックから出るときのことを考えると明らかになる．もし，STACKTOP がただ1個で，大域的なものとすると，ブロックから出るたびに，そのブロックの中で宣言されている配列に使われた場所をスタックから取除くのでこの変数の値を変更しなければならない．こうすると，いくつかのブロックを一度に飛越すような命令があると面倒が生ずる．ところがここで用いているようなシステムでは，規則 (8.9.2) によって，変更は不要である．したがって，ブロックからの正常な出口ではなにもする必要がないのである．

このブロック構造処理の例を示そう．図 8.8 **a** は，図 8.7 の手続き A が呼出され，初期設定された直後のスタックの内容を示している.[†] 名札 L1 で実際の実行を開始する前の状態である．図 8.8 **b** は，手続きの主ブロックに入った直後で，配列宣言処理を行う前の状態のスタックの内容を示している．つぎに図 8.8 **c** では，配列が割付けられ，ブロック 1 用の STACKTOP 変数が変化してスタックの最新の先頭を指すようになる．手続き A 用の STACKTOP の内容はもとのままであることに注意．ブロック 2 に入ると，ブロック 1 用の STACKTOP 変数の内容がブロック 2 用のそれに代入される（図 8.8 **d**）．ここでは，配列の宣言がないので，これ以上の処理は不要．

ブロック 2 の出口では，もはやブロック 2 の変数は活きていないので図 8.8 **c** に戻す，ここでは（ブロックの出口では），何の命令も実行されていないことに注意．変数 D, E や b2 用の STACKTOP はなお，記憶装置の上に残ってはいるが，もはや参照する手掛りがないのである．

ブロック 3 に入ると，STACKTOP 変数の値を写しとり，配列 A の記憶場所

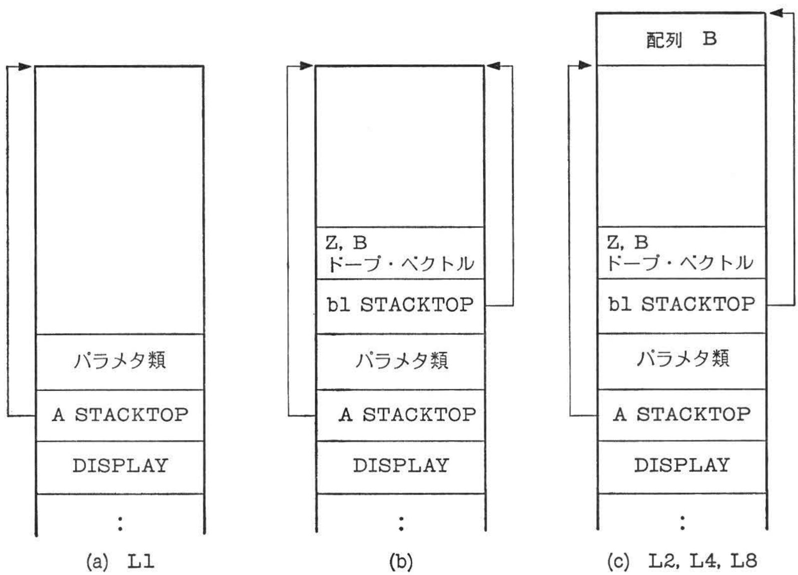


図 8.8 ブロック構造処理

† **訳注** 227 ページ図 8.7 の上の表と図 8.8 ではスタックの上下が逆になっているので要注意.

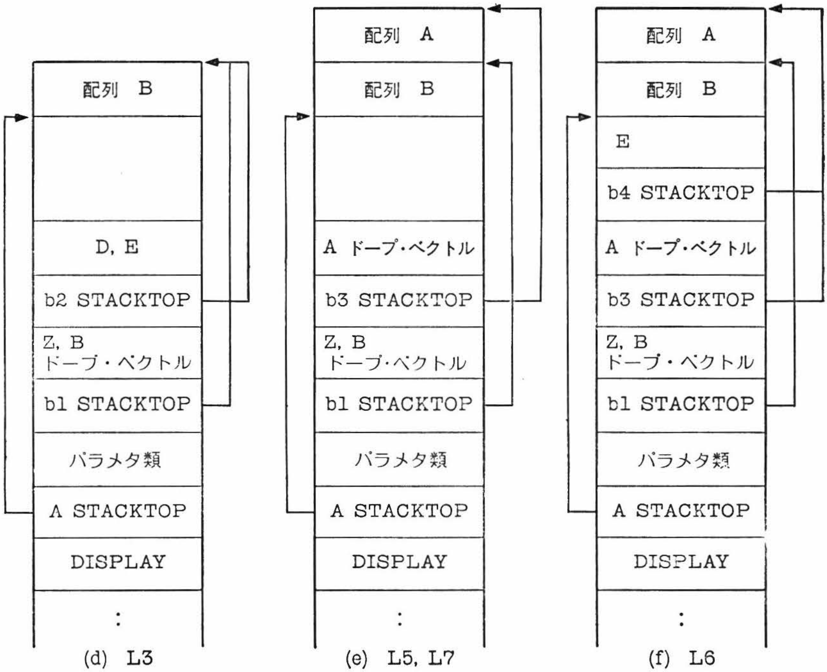


図 8.8　ブロック構造処理（つづき）

を割付けて図8.8**e**が得られる．ここでは，STACKTOPが3個現れる．すなわち，スタックの先頭を指しているブロック3（活きているブロック）のもの，ブロック1のもの，それに手続き自身でもあるブロック0のものである．

ブロック4に入ると，再びSTACKTOPを写しとり，図8.8**f**を得る．このあとは，ブロックからの脱出となるので，ブロック4を出て，図8.8**e**，ブロック3から出て図8.8**c**，ブロック1から出て図8.8**a**のようにもとに戻ってくる．

手続き呼出し

手続き呼出しのための命令群は，名前呼びの引数のためのサンク(thunk)ルーチン，引数の一覧表を組上げる一連の命令，そして，実際に手続きに制御を移す命令からなっている．サンク・ルーチンの形式は後述する．DISPLAYのところで述べたように，引数の一覧表は，実引数と下記の四つのかげのパラメタからなっている：

1. 戻り先番地
2. 実引数 DISPLAY の番地
3. 大域的 DISPLAY の番地
4. 現在の STACKTOP の内容（この手続きを呼んだ時のスタックの先頭番地）

第二のかげのパラメタの値は手続き呼出しがあった時点での ACTIVEAREA 上にある値であり，呼出しの実引数はそこに現れている．このパラメタはとっておいて，サンク・ルーチンが，その参照すべき変数のあり場所を知る助けにするために使用される．手続きの出口で，呼出した時の DISPLAY を再現するために，この値を再び ACTIVEAREA に設定し直す．

ここでは，呼出される手続き名としては，手続きの名と指定された仮引数ではないものとしよう．この制限をつけると，第二，第三のかげのパラメタは，同じものになる．第三のパラメタ，大域的 DISPLAY の番地というのは，呼ばれた手続きが，自分の DISPLAY を構成するために使われる．これを説明しよう．

手続き A を手続き B(あるいは主プログラム)から呼出すとしよう．するとつぎの 2 通りの場合が考えられる．第一は，手続き A が手続き B の内部で宣言されている場合である(図 8.9 **a**)．B が水準 i にあるならば，A は水準 $i+1$ となる．A は B にとって大域的な全ての変数と，A の宣言を囲む B のブロック内で宣言された任意の変数を参照することができる．したがって A の DISPLAY を組立てるには，A は，この第三のパラメタによって与えられた番地（大域的 DISPLAY の番地）から始まる $i+1$ 個の記憶場所を，写しとるだけでよい．

第二の場合（図 8.9 **b**）は，A は B の外で宣言されているが，B から参照可能な関係にある場合である．すなわち，A は B を囲むブロックの中で宣言されている．もし A が水準 $i+1$ にあれば，A と B はともに水準 $0, 1, \cdots, i$ にある各手続き中の変数を参照することができる．第一の場合のように A は，大域的 DISPLAY の番地から始まる $i+1$ 個の場所を写しとるだけでよい．

4 番目のかげのパラメタ（現在の STACKTOP の値）は，実行時のスタックの現在の最上段の場所の番地である．これは，呼ばれた手続きが，それに対応する記憶場所をどこに積上げたらよいかを知るために使われる．手続き呼出しがサンクの中で発生したのでなければ，† この値が，呼出しの生じたブロックの

---

† **訳注** サンクの中で手続き呼出しが発生する場合については 234 ページ参照．

```
PROCEDURE B;
  BEGIN PROCEDURE A;
          BEGIN…END;
L1: A;
  END;
```

(a)

```
PROCEDURE A;
  BEGIN…END;
PROCEDURE B;
  BEGIN…L1: A;…END;
      :
```

(b)

図 8.9　手続きAを呼出している手続きB

STACKTOP 変数にある値である.

手続きの初期設定と手続きからの脱出処理

手続きが呼ばれると，そのデータ領域のために記憶場所を確保し，その手続き用のDISPLAYを初期設定し，かげのパラメタと実引数をそのデータ領域の中に写しとらなければならない．水準$i+1$にある手続きが行う処理の手順の詳細はつぎのとおりである

1. 手続きのデータ領域用の記憶場所をスタックにのせる.（4番目のかげのパラメタが，その時点でのスタックの先頭番地を指している.）このデータ領域の先頭番地をデータ領域の$i+1$番目の場所（すなわち DISPLAY の$i+1$番目）にしまい，それを大域的な ACTIVEAREA に置く．そして，この手続きに固有な STACKTOP 変数に新しいスタックの先頭番地をしまう.
2. 4個のかげのパラメタをこのデータ領域に移す.
3. 3番目のかげのパラメタによってきまる番地の最初から$i+1$個の番地の内容をデータ領域の0, 1, ···, i番目の場所にそれぞれ移す.
4. 実引数をつぎのようにして移す.
   a） 配列名：ドープ・ベクトルを写しとる.
   b） 手続き名：実引数の番地を写しとる（これの使用法は後述）
   c） 名前呼びの実引数：対応するサンクの実行開始番地を写しとる.
   d） 値呼び実引数：実引数の番地にある値を新しいデータ領域に移す.

手続きからの脱出では，実行の環境を呼出し時に戻してやらなければならない．（その上，もし手続きが関数であれば，関数値をレジスタにのせて返してやる必要がある.）環境をもとに戻すのは容易である．すなわち，大域的な ACTIVEAREA に，手続きに渡された2番目のかげのパラメタをそのままおいてやればよい．スタックの先頭番地がどこかについては何もする必要がない.

手続きのデータ領域がもう使用されなくなるのだから，先頭番地の位置が必要と考えるかもしれないが，先頭番地の位置を示す STACKTOP が，規則（8.9.2）によって，現在活きているブロックによって定められ，これが今変ったのである．

サンクとその呼出し方法

8.7 節に述べたように，サンクは，名前呼びの仮引数に対応する実引数の番地を計算して返してよこすルーチンである．実引数は式であるから，宣言もブロックも含んでいない．したがって，一時的な結果の記憶場所としては，呼出しのあったブロックに属する記憶場所を使うことができる．サンクの中に現れる式の中に1次子として関数呼出しがあれば，これには特別な考慮を払わなければならない．これを以下に述べる．その前に，サンクの呼出しとサンクの構成を調べよう．

手続き中で名前呼びの仮引数 X を参照するたびに，サンクの呼出しが生ずる．その際引数一覧表はつぎのようになる：

1. 戻り先番地.
2. ACTIVEAREA の現在の値.
3. X を仮引数とする手続きを呼出したときの第二のかげのパラメタの値.
4. スタックの現在の最上段の番地.

これは，見てのとおり，引数を持たない手続きの呼出しの場合の引数一覧表に非常によく似ている．第二のパラメタは，サンクの中では直接は使われない．これは一たん退避され，サンクから戻るときにもとに戻されるだけである．第三のパラメタはサンクの環境を初期設定するために使用される．通常の手続き呼出しでの引数一覧表を一見すれば，この第三のパラメタは呼出し側の手続きの DISPLAY の番地であることがすぐわかるだろう．4番目のパラメタは，サンクが関数を呼んでいる場合に，呼出しのかげのパラメタとして，スタックの先頭番地の位置をその関数に知らせるために，サンクが用いるためのものである．サンク自身は，以下の処理を行う：

1. 第三のパラメタを ACTIVEAREA におく.
2. 4個のパラメタを全て（ACTIVEAREA によって参照されるデータ領域の中にある）一時的な結果の記憶場所[†] にしまう.

---

† **訳注**　この場所はコンパイル時に名前呼びの実引数に対するサンクを作るときに決る.

3. 実引数の番地を計算する.（これがサンクの本体である.）
4. サンク呼出しの第二のかげのパラメタをACTIVEAREAに置く.
5. 手順3で計算した番地を，特定のレジスタに置いて，戻り先番地に戻る.

実引数を参照するためにサンクによって，特定のレジスタにおかれた番地が，このサンクを呼んだ手続きによって用いられる.

サンク中の関数呼出しは4番目のかげのパラメタだけを除けば通常のとおりに行われる. このパラメタは，スタックの現在の最上段の番地と考えられており，もしサンクの中でなければ，関数呼出しの生じたブロックに固有なSTACKTOPの値である. しかし，関数呼出しがサンクの中で生じると，スタックの現在の最上段は，サンクの呼出しが現れたブロックのSTACKTOPで与えられる.† これがサンク呼出しの4番目のかげのパラメタの役割である.

手続き名を実引数として用いた場合

ここでは，現実のプログラムでは，滅多に起らない非常に奇妙な場合に対処するための問題を提示する. まれにしか起らないといっても，それを処理することができなければならない. 図8.5(223ページ参照）のプログラムを考えよう. これは，それぞれ入れ子になった2組の並行な手続きからなっている. このグラムを慎重に実行してみよう. まず名札L1で，手続きCが呼ばれる. Cの中の名札L2で手続きAが呼ばれる. このとき，手続きの名前Dが実引数として用いられている. 手続きAは，名札L3でBを呼出す. このときの実引数は，Aの仮引数であるXである. 最後に名札L4で手続きZを呼出す(Zは手続きBの仮引数). Bの仮引数Zは，Xに対応し，それはさらにDであったから，手続きDが呼出されることになる. ここでDは，二つのことを知っていなければならない：

1. 名札L4で活きているDISPLAYの番地. これによって，L4での手続き呼出しの際の実引数に対するサンクにこの番地を与えることができる. そこでこれを**実引数**DISPLAYの番地と呼ぶ.
2. Dを参照した時点での手続き（ここではC）のDISPLAYの番地. これによって，Dは，大域的な変数を参照することができる. そこで，これを

† 訳注　サンクの中からはこの値を直接参照できないので4番目のかげのパラメタとして渡しておかねばならない.

大域的 DISPLAY の番地と呼ぶ.

これらがそれぞれ, D の第二, 第三のパラメータである. 普通の手続き呼出しでは, 両パラメータは, 同一であって, 呼出し点における ACTIVEAREA の内容そのものである.

実引数として 手続き 名が現れると, 引数一覧表で 渡されるのは二つの情報

手続きの実行開始番地
呼出し点における ACVITEAREA の値

の入った表の番地である. 後者の値が, 第三のかげのパラメタ (大域的 DISPLAY の番地) として, 仮引数に対応する手続を呼出したときに, 用いられる. 仮引数となっている手続きがそれ自身実引数となる場合は, 実引数は, その手続きに渡された実引数の番地で表される.

これを説明するために, 図 8.5 のプログラムの名札 L1, L2, L3, L4 における呼出しの際の引数一覧表を以下に示した. L2 では, DAC という番地が実引数 D を記述していることに注意しよう. この番地は, L3 における呼出しで, その 引数一覧表に 写しとられる. そして, 手続き B 中の L4 における呼出しで, 大域的 DISPLAY の番地を与えるために用いられている.

L1 : 手続き C の呼出し

1. 戻り先番地
2. 主プログラムのデータ領域の番地
3. 主プログラムのデータ領域の番地
4. 呼出し点の STACKTOP の値

L2 : 手続き A の呼出し

1. 戻り先番地
2. 手続き C のデータ領域の番地
3. 手続き C のデータ領域の番地
4. 呼出し点の STACKTOP の値
5. DAC の番地 (下を見よ)
6. 実引数 "1" 用のサンクの番地

L3 : 手続き B の呼出し

1. 戻り先番地
2. 手続き A のデータ領域の番地
3. 手続き A のデータ領域の番地
4. 呼出し点の STACKTOP の値
5. DAC の番地 (下を見よ)

L4 : 手続き Z (≡D) の呼出し

1. 戻り先番地
2. 手続き B のデータ領域の番地
3. 手続き C のデータ領域の番地
4. 呼出し点の STACKTOP の値
5. "F[1] + Y" 用のサンクの番地

DAC	
	手続きDの番地
	手続きCのデータ領域の番地

分　　岐

GOTO Lに対して，名札Lが手続きの内部にあれば，単純な分岐命令を作り出す以外なにもしなくてよい．しかしながら，Lがその手続きを囲む，水準 i にある手続き中にある場合はどうであろうか．その手続きのデータ領域の先頭番地を ACTIVEAREA にのせてから分岐を行わなければならない．この番地は，活きている DISPLAY の中にあるので

$$\mathrm{ACTIVEAREA} := \mathrm{CONTENTS}\,(\mathrm{ACTIVEAREA} + i)$$

によって簡単に実現できる．Lが仮引数の場合は，引数一覧表中の対応する番地は，サンクの番地であるから，このサンクを普通どおりに呼出す．名札（または行先式）が実引数である場合のサンクは，呼出し点に戻ってこないで，初期設定を行った後，サンクで計算した番地にそのまま分岐してしまう．

この特別な場合でも，手続きやブロックから脱出するときに，記憶域を解放することには何も気を配らなくてよいことを再度注意しておく．スタックの先頭は，つねに，活きているブロックの STACKTOP によって与えられているからである．

演習問題 8.9

1. 名前呼びの実引数を，呼出しのあったブロックで宣言された，引数のないポインタ関数として取扱う方法の可能性を論じてみよ．この関数は，実引数の番地を関数値とする．すると，実引数も仮引数も，手続き名となり，手続きの本体で，仮引数を参照することが関数呼出しに対応する．この方法を，われわれの方法と比べてみよ．かげのパラメタは，この方法でも，全て必要か．
2. より一般的な記憶割付の機構を使用することを考えよう．大きさ SIZE のデータ領域を割当てるために，GETAREA (ADDRESS, SIZE) を，ADDRESS 番地から始まる SIZE 個の大きさのデータ領域を解放するために，FREEAREA (ADDRESS, SIZE) をそれぞれ呼出さねばならないものとする．GETAREA も FREEAREA も，必要な始末を全て自分でや

るので，STACKTOPは不要になる．したがって，手続き呼出しの際にかかげのパラメタは三つでよい．このとき，ブロックの開設，ブロックの閉鎖，手続きの開始，手続きの終了，そしてGOTO文の処理を詳しく述べよ．ブロックの閉鎖と，手続きの終了は注意深く行うこと．特にブロックや手続きの境界をまたぐようなGOTO文の処理には注意を払うこと．このときには，いくつものデータ領域を解放する必要がありうるからである．

3. 問題2で，GETAREAとFREEAREAが，記憶割付にスタックの機構を用いることができるように，配列と手続きのデータ領域を解放する順序にいくつか制限をつけてみよ．
4. ALGOLにおいて，各ブロックや手続きの一つの実行に対して一つのデータ領域を割当てる記憶管理法をほかに設計してみよ．すると各データ領域に対するDISPLAYは，それを囲むブロックや手続きのデータ領域の番地を保持しなければならない．
5. つねに，参照可能なデータ領域の全ての番地をのせている大域的なDISPLAYをただ一つ持つような記憶割付けの機構を設計せよ．（データ領域は，手続きだけに割当てるか，ブロックと手続きの両方に割当るかは，どちらでもよいから，自分で決めること．）手続きを呼出すときは，注意が必要である．すなわち，大域的なDISPLAYの一部を，手続きのデータ領域の中に退避しておき，手続きから戻るときに，回復する必要がおこるかもしれない．サンクを呼出すときにも，もっとややこしい処理がいる．

8.10 動的な記憶割付

これまで見て来たように，実行時において動的な記憶割付を必要とする言語がある．そこでは，一連の記憶場所が確保され，使用され，そして解放されて次の使用に備える．8.9節に述べたALGOLの実現では，記憶割付は，単一のスタックの機構を用いた．いろいろの応用の中には，記憶場所の割当や解放を標準的に，あるいは前もってきめた順序では，遂行できないので，スタックを使用できないものもある．本章では二つの例を見てきた：一つはXPLにおける文字列への記憶割付であり（8.5節），一つはALGOL Wの，構造を持ったデータの記憶割付である（8.6節）．PL/Iもこの例であり，PL/Iでは，プログラマが，記憶場所の確保や解放が必要なときに，自分でそれぞれALLOCATE文やFREE文を使って，きめることができる．

一般に記憶割付には主要な方法が2通りある．両者とも，GETAREA (ADDRESS, SIZE) ルーチンに類するルーチンを呼んで SIZE 個の長さの領域を割付けそしてこの領域の先頭番地を ADDRESS に与える．

第一の方法では，確保された記憶場所は FREEAREA (ADDRESS, SIZE) ルーチンを呼出して，陽に解放しなければならない．一方，第二の方法は，記憶場所を解放することを陽には行わない．その代り，GETAREA ルーチンを呼んで，必要な大きさの領域がなくなっていると，GETAREA が FREEGARBAGE ルーチンを呼んで，このプログラムがもはや使用していない場所を主記憶から拾い集めてきてシステムに返してやる．さらに，このルーチンは，使用中の領域をつめ合せて，その結果，未使用領域を再び一つながりになるようにすることもできる．

まず，第一の方法を実施する技法の一つを説明しよう．

境界目印法による記憶割付

記憶割付はつぎのように進行する．プログラムが実行を開始すると，まず，大きな一連の記憶場所が自由（未使用）領域となる．プログラムの実行につれて，GETAREA ルーチンが何度か呼ばれることになる．このルーチンは，そのたびに，自由領域の先頭から必要なだけの記憶場所を切取って割付けるのでつぎのような形になる：

使用中	使用中	…	使用中	自由（未使用）

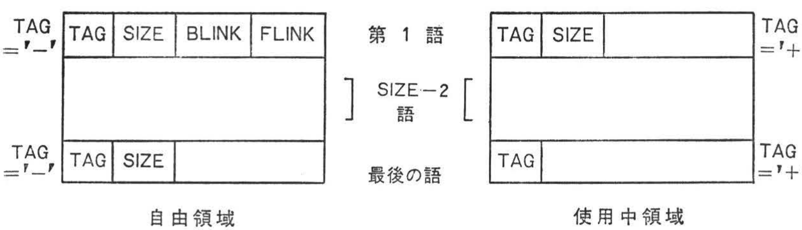
ここで使用中領域の大きさは，必ずしも同一ではない．時には使用中の領域を解放するために FREEAREA が呼ばれることがあろう．解放される領域は，一般には最後の使用領域ではない，このように GETAREA と FREEAREA を何度か呼ぶうちに，領域がつぎのような状況になるとしよう：

使用中	…	使用中	自　由	使用中	…	使用中	自　由	自　由	使用中	自　由

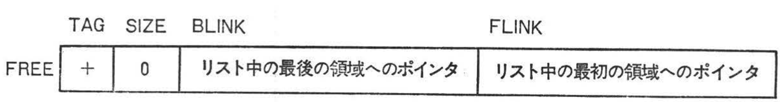
ここで，全ての領域が同じ大きさというわけではないことを再度注意する．システムは全ての自由領域が再び使用できるようにそれらの情報を保持しておかなければならない．さらに隣合った自由領域は一つの自由領域にまとめて，自由領域が細分化されすぎて使いにくくならないようにしなければならない．

ここで述べる特別の方法は，Knuth (68) の**境界目印法** (boundary tag method) である．この方法では，システム用として，各領域中に，領域の両端に場所を二つ確保しておくことが必要である．場所を二つとることは，そんなに高価なものではない．なんとなれば，この方法が使われるような状況では領域は手続きのデータ領域や，配列のように比較的大きなものを扱っているからである．領域を解放したり，隣合った自由領域をできるかぎり一つにしたりするのに要する時間が本質的に一定であることがこの方法の利点である．他の方法では，自由領域の処理に，何らかのリストの探索をしなければならないのである．

使用中領域および未使用領域の形式は下図のようになっている．領域の第1語には，TAG 欄があって，その領域が使用中か自由かを示している．また SIZE 欄は，領域の大きさ（語数）を示す数が入る．自由領域はそれぞれ双方向リストとして結合されており FLINK 欄（前進リンク）は，リスト上での次の領域を示す値が入り，BLINK 欄（後退リンク）には直前の領域へのポインタが入る．



さらに，つぎのような形式の変数 FREE が用意される：



リスト中の最初の領域の BLINK 欄には FREE 変数の番地がポインタとして入る．一方最後の領域の FLINK 欄も FREE 変数の番地を指している．最後に使用中の一かたまりの記憶域の第1語と最後の語の TAG 欄には，それが使用中であることを示す‘＋’が入っている．このようをしておくと，隣合っ

た領域を一つながりにする処理を簡単化することができる.

いよいよGETAREAルーチンを与える用意が整った. このルーチンは“第一適合”法を使っている. すなわち, 自由領域のリストを先頭からたぐって, 最初に見付かった十分な大きさのある領域を取ってくる. 一方“最良適合”法の方が一見してよいように思えるが, 必ずしもそうでないし (Kunth(68)を参照せよ), 時間が余計にいることははっきりしている. このルーチンでは, SIZE語の大きさの領域が要求されると, SIZE + 10 語までの大きさの領域が見つかれば, ここで余りを分割して自由領域に残すようなことはしない. もしそうすれば, 小さな, 未使用の領域が非常にたくさんばらまかれてしまうことになるからである. ここで 10 という数は, 適宜に選んだものである.

PROCEDURE GETAREA(ADDRESS, SIZE);

BEGIN POINTER P, Q, Q1; INTEGER S, S1;	Pは現在の領域へのポインタ.
P:=FREE. FLINK;	Sは必要な大きさの最小値. S1
S:=SIZE+2; S1:=SIZE+10;	より小さい領域は二つの領域に
	分割しない.
WHILE P ≠ ADDRESS(FREE) DO	領域を順に見る.
BEGIN IF P. SIZE ≧ S1 THEN	もし領域の大きさがS1以上あ
BEGIN S1:=P. SIZE−S; Q:=P+S;	れば, 領域をS語とS1語にわ
Q1:=Q+S1−1;	ける.
CONTENTS(Q):=CONTENTS(P);	Pがポイントする語の内容を全
Q. SIZE:=Q1. SIZE:=S1; Q. TAG:='−';	てQがポイントする語に移す.
P. FLINK. BLINK:=P. BLINK. FLINK:=Q;	下部の領域(Q)を自由領域リス
Q:=Q−1; P. SIZE:=S; GOTO FOUND;	トにつなぎ直し上部(P)の
END;	SIZEを調整して脱出.
IF P. SIZE ≧ S AND P. SIZE ≦ S1 THEN	もし領域がちょうどよい大きさ
BEGIN P. BLINK. FLINK:=P. FLINK;	ならば, それを自由領域のリス
P. FLINK. BLINK:=P. BLINK;	トからはずす.
Q:=P+P. SIZE−1; GOTO FOUND	
END;	
P:=P. FLINK	領域が小さければ, 次の領域を
END;	試す.
WRITE('ERROR. NO BLOCK.'); STOP;	ちょうどよい大きさの領域はな
FOUND: P. TAG:=Q. TAG:='+';	い.
ADDRESS:=P+1;	領域はP番地からQ番地の間
END	にある.

FREEAREA ルーチンは領域を解放し, 可能ならば隣合った自由領域を一つにまとめる. この方法では, 引数SIZEの値は利用していない. それは, 必要な情報は, 領域自身の中に入っているからである.

PROCEDURE FREEAREA(ADDRESS, SIZE);

BEGIN POINTER L, M, U, Q; M:=ADDRESS−1; U:=M+M. SIZE; L:=M−1;	Mをこの領域へのポインタとする．Uはこの領域のうしろに接する領域へのポインタ．Lは，この領域の直前に接する領域の最後尾．もし直後の領域が自由ならば，それをリストからはずし，合併する．
IF U. TAG ='−' THEN BEGIN M. SIZE:=M. SIZE+U. SIZE; U. BLINK. FLINK:=U. FLINK; U. FLINK. BLINK:=U. BLINK; END;	
Q:=M+M. SIZE−1; IF L. TAG = '−' THEN BEGIN L:=M−L. SIZE; L. SIZE:=L. SIZE+M. SIZE; Q. SIZE:=L. SIZE; Q. TAG:='−' END	QはMの最後尾をポイント．もし直前の領域が自由ならばMの領域と合併する．
ELSE BEGIN Q. SIZE:= M. SIZE; M. TAG:=Q. TAG:='−'; M. FLINK:=FREE. FLINK; M. BLINK:=ADDRESS(FREE); FREE. FLINK. BLINK:=M; FREE. FLINK:=M END END	さもなければ，この領域を自由領域のリストの先頭につなぐ．

くず拾い (Garbage Collection)

記憶割付の第二の方法では，GETAREA が適当な領域を発見できないときは，FREEGARBAGE ルーチンを呼び，それが，未使用の領域を見つけ，自由領域リストにつなぐ．これを行うために，FREEGARBAGE ルーチンはつぎの諸点を決定できなければならない：

1. 宣言された全てのポインタ変数の記憶場所．
2. 各ポインタが指している場所，領域の長さ，それがさらに別のポインタを含んでいるかどうかに関する確実な情報．
3. ポインタによって参照されているデータの中にある全てのポインタに対してはそれぞれの長さ，記憶場所に関する情報．

これは，大変やっかいな注文であり，これに応えるには，ポインタをおとなしく使わなければならないことが容易に想像できよう．このために，くず拾いの方法は，普通，ポインタが正しく使われ，データの形式の種類が少いことが確

実なシステムで使用される．LISP は，そのようなシステムの好例である．別の例として，XPL の文字列処理の機構がある．これには，ポインタによって参照されるのが，文字列だけであるという利点がある．つまり，指されたデータの中には，別のポインタを決して含まないのである．

さらにむずかしいのは，任意の時点で，どの領域が自由であるかを確実に決定する問題である．つぎのような ALGOL W の例でこれをうまく説明しよう：

```
RECORD LISTOFREAL(REAL A;POINTER NEXT);
P:= LISTOFREAL(2.0, LISTOFREAL(3.0,0));
```

上の文を実行するには，まずLISTOFREALデータに記憶場所を割付けこの場所を（3.0，0）と初期設定する．つぎに，第二の LISTOFREA データに記憶場所を割付け，このデータに（2.0，Q）と初期値を与える．ただしQは最初のデータ（3.0，0）を指示するポインタである．ところで，第二のデータ用の場所が存在しないとしよう．そこで FREEGARBAGE が呼出される．しかしながら番地Qはポインタ変数の中にはおさまっていないので（3.0，0）を入れる記憶場所は未使用のようにみえる．

このことは，このような状況が訂正できないものだというのではなくちょっと不注意でいると思いがけずこのようなことになるということを示している．

くず拾いルーチンの進行は通常2段階に分かれる．第1段は適当な方法で使用されているデータに全て目印をつけるもので，普通の方法では，目印をつけるためだけに使用する余分のビットを各データが持っているが，場合によっては面倒が生じることがある．これと対比される方法では，目印をつけるビットを全て一つの表に集めて，データとそのビットを適当に対応させておく．しかし，この方法では，目印用のビットのために特別な表が必要で，くず拾いルーチンが呼出された時，ほんのわずかな自由領域しか存在しない恐れがある．

第2段では，領域の中を順にたどって，目印のついていない領域を自由領域のリストの方に置き，同時に目印を消す．これは，再度くず拾いルーチンが呼出される時にまた正しく実行できるための初期設定である．

さらに3段目が加わって，領域をきちんとつめ合せて，自由領域全部で一つの大きなブロックにするような場合もある．このためには，データを移動させる際にポインタの値を変更する処置をくず拾いルーチンに行わせる必要がある．

くず拾いの各種アルゴリズムについては Knuth (68) の第2章を参照され

たい.

2層式割付システム

記憶割付を2層にして行っているシステムもある．適当な方法で，主記憶の大きなブロックをいくつか確保して使用し，各ブロックは，別の方法を用いて区分される．ALGOL W はこの方法を用いている．それぞれの構造体ごとに一意的な整数値を対応させ，それぞれに対して表の中にテンプレートを一つとっておく．このテンプレートには1個の構造化されたデータが用いる記憶場所の個数とそのデータ中にポインタがあればその記憶場所とが含まれている．現実に構造化されたデータは (IBM 360 では) 1バイトを余分に持ち，このうち7ビットで構造体の型を示す整数値を表し，残りの1ビットをくず拾いの目印のために用いている．

それぞれ異なった構造体の型が必要に応じていつでも主記憶上の別の大ブロックに割付けられる．そのブロックの中に，その型を持ったデータが全てたくわえられ，自由領域は自由領域のリストに連らねておく．これによって構造化されたデータへの記憶場所の割付は，その型の自由領域リストの先頭の場所を取ってくることによって能率よく行うことができるようになる．

構造体のある型に対する自由領域リストがなくなると，システムはさらに別の大ブロックを要求する．これ以上大ブロックがなくなるとくず拾いルーチンが働いて，それぞれの型ごとに未使用のデータを自由領域リストに返してやり，空となっている大ブロックが主記憶割付ルーチンに返却される．

8.11 参考文献概説

配列の参照と配置の方法に関する短い概説が二つある．Hellerman (62) と Gower (62) のものである．実際，これらの方法は多くのプログラマが，自分の仕事の中で，独立に繰返し見付けだしてきたもので，だれの仕事とするのは困難である．Knuth (68) は，配列の配置について，他にいくつかの方法を与えており，特に三角行列や粗行列のものは注目に値する．

いろいろな型のデータ構造の実現法に関する議論には，Wirth, Hoare (66); Standish (68); Bauer, Becker, Graham (68); IBM の PL/I ロジックマニュアルがある．Hoare (68) は，データ構造とその実現法に関して，より完備した議論を与えている．

CACM の1961年1月号は，ALGOL の作成に関するもので歴史的な観点か

ら読む価値がある．もっとも，ここに書かれている技法の多くは時代遅れとなっている．Randell, Russell (64) は ALGOL 作成技法について書いた．その本の中で，一つのブロックの実行のたびに一つのデータ領域を，大域的 DISPLAY を用いながら，割付ける方法を述べている．8.9 節で詳しく述べた特別な技法は ALCOR ILLINOIS 7090 コンパイラ (Gries, Paul, Wiehle (65)) で用いられたものである．

何度もふれたように，Knuth (68) の第 2 章は，記憶割付技法のすばらしい解説を含んでおり，さらに多くの技法が練習問題に入っている．Kunth は，さらに，いくつかの方法の能率も解析している．Ross (67 b) は AED 自由領域管理パッケージを論じている．これは，一般的なシステムにおける 2 層記憶割付機構の好例である．

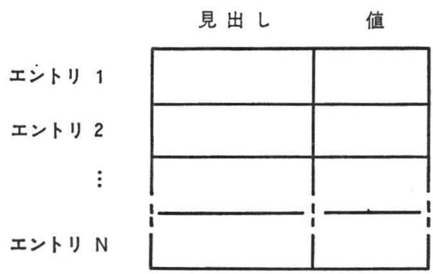
第 9 章

記号表の作成

原始プログラムの意味的な正しさを調べて目的コードを生成するには，その中に用いられている**名前** (identifier) の属性を知ることが必要である．これらの属性は宣言の中やプログラム中での名前の使われ方からわかるもので，これらの情報を集めて**記号表** (symbol table) の中にしまう．本章は記号表の構成，組立て，探索の方法を議論する．一般的な表を構成する問題の基礎にも興味があるが，ここでの関心は記号表という特定の問題に集中することにする．

9.1 序　　論

どんな表もつぎのような一般的な形式を有している．



ここで，左側が見出しの一覧で，右側がそれに対応する値である．各エントリは，計算機の複数個の記憶単位を占めるのが普通である．もし一つのエントリが k 語を占めるとすれば，N エントリでは，$k \times N$ 語の記憶場所を必要とする．これを実現する方法は2通りある．

1. ひき続く k 語を各エントリとし $k \times N$ 語の表を持つ．

2. それぞれ N 語からなる $T_1, T_2, \cdots, T_k$ という k 個の表を作る．i 番目のエントリは $T_1[i], \cdots, T_k[i]$ にある．

これら 2 通りの方法の違いは，プログラム作成上の便宜によるにすぎない．

コンパイラの場合，表の見出しは，記号または名前であり，値は，それらの属性である．名前を構成する文字数は可変なので，見出し部には，名前そのものを置くかわりに，名前へのポインタを置くことができる．一方名前は，名前用の特別なリストにたくわえる．名前を構成する文字数は見出しの一部として表に置くこともできるし，名前用のリストの方に，名前の直前に置くこともできる．図 9.1 は，名前 I，MAX，J に対するエントリを持つ表で，上の 2 通りの技法の違いを説明したものである．

コンパイラが原始プログラムの翻訳を開始する時点では，記号表は空であるか，いくつかの予約語や標準関数の名前がのっているだけである．コンパイルが進むにつれて，新しい名前に会うごとにその名前のエントリを表に加える．これは一度だけで，以後その名前が現れるたびに一度ずつ表引きが行われる．コンパイル時間の多くはこの表引きの時間に費やされるので能率のよい表引きができるように表を構成する方法を研究することは重要である．

種々の表引きの方法についてその表引きに要する時間を比較してみたい．これはある特定の名前を見出すのに要する見出しとの比較回数の期待値 E で行ってみよう．この期待値は，任意の時点における**表占有率** (load factor) lf

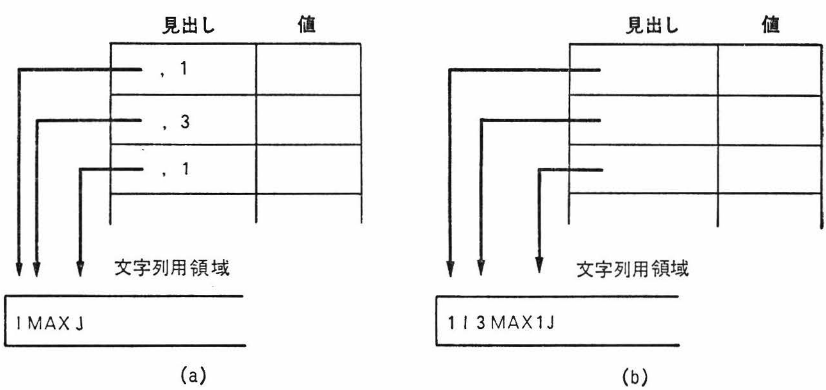


図 9.1　見出しの格納法

――すなわち，現在のエントリ数 n を，表の最大エントリ数 N で割った値

$$lf = n/N$$

によることが多い．以下で，比較回数の期待値の公式を示すが，その導出はしない．実際の解析は，9.6 節に挙げる文献を参照されたい．

9.2　整列済みの表と未整列の表

表を作り上げる最も容易な方法は，名前の出現順にその見出し用のエントリを追加してゆく方法で，並べかえなどはしない．表引きを行う際は，当該のものが見付かるまで，全てのエントリと比較することが必要である．n 個エントリのある表では比較の回数の平均は $n/2$ である．n が大きくなる（20 以上）と，この方法は能率が悪い．もし表のエントリが，見出しの持っている適当な順序に応じて並んでいれば，より効率的な探索を行うことができる．コンパイラでは，見出しは，文字列なので，文字列の機械の内部表現を数とみて順序付けるのが最も自然である．この順序は，辞書式順序になるのが普通である．すなわち，A，AB，ABC，AC，BB という文字列は，順に並んでいることになる．順に並んだ n 個のエントリを持つテーブルを効率よく探索する方法の一つに，**二分探索法** (binary search) あるいは**対数的探索法** (logarithmic search) がある．探索の対象となる記号 S を，中央のエントリ $(n+1)/2$ の見出しと比較する．もしこのエントリが，求めるものでなければ，S が比較の対象の見出しよりも小さいか大きいかに応じて 1 番から $(n+1)/2-1$ 番のエントリの中か，$(n+1)/2+1$ 番から n 番のエントリのいずれかで，探索を行なえばよい．このようにして，次々と探索の対象ブロックを半分にして表引きを繰返す．各段で，S を含む可能性のあるエントリの数は 1/2 になってゆくので，比較回数の最大は $1+\log_2 n$ である．

もし $n=2$ であれば，比較は高々 2 回である．$n=3$ ならば，3 回，$n=8$ ならば 4 回である．$n=128$ ならば，二分探索法では，高々 8 回の比較に対して，順序づけを行わない方法では，平均 64 回の比較が必要である．二分探索法のプログラムは下記のとおりである．

PROCEDURE binsearch(T, n, ARG, k); STRING ARRAY T; STRING ARG; INTEGER n, k; BEGIN INTEGER i, j; i:=1; j:=n; LOOP: k:=(i+j)//2; IF ARG ≠ T[k] THEN BEGIN IF ARG < T[k] THEN j:=k−1 ELSE i:=k+1; IF j ≧ i THEN GOTO LOOP; k:=0; END END	配列Tの中にARGに等しい要素を見出す．見つかれば，エントリの番号をkに与える，さもなければk＝0．n＞0とする．最初，表引きの対象は配列の全要素．kはブロックの中央のエントリを指す．ARGがエントリkに等しいか調べる． ARGがブロックの上半か下半かのいずれに属するか調べ，調べるべきブロックの大きさが0になっていたら，ARGはこの配列にない．

二分探索法を用いるためには，エントリは，整列済みでなければならない．そのための方法の一つとして，**挿入整列法** (sort by insertion) がある．この方法が使用される理由は，コンパイラでは表が最終的に完成される以前に，表引きが行われるので，いつでも整列された状態になっている必要があるからである．この方法では，記号 S のエントリはつぎのようにして挿入される．

1. 手続き binsearch を用いて $T[k] < S < T[k+1]$ となる k を見付ける.†
2. エントリ n をエントリ $n+1$ に，エントリ $n-1$ をエントリ n に，$\cdots$，エントリ $k+1$ を $k+2$ に移す．
3. S をエントリ $k+1$ に置く（この場所はステップ2で空にされた）．

もし何らかの表引きを行う前に，表が完全に一ぱいになっていると，エントリを全て集めてしまうまで整列の実施を延ばす可能性がある．ここでは整列技法については議論しない．9.6 節に参考文献を挙げた．

9.3　ハッシュ技法

これは，記号を表中のエントリの番号に変換する方法である（表は N 個のエントリからなり，$0, 1, \cdots, N-1$ と番号づけられているものとする）．この番号は，記号をハッシュする——すなわち，記号に（ときにはその長さに）簡単な数値演算または論理演算を施す——ことによって得られる．

† **訳注**　表に ARG が見付からない場合，さらに $S < T[1]$，$T[n] < S$ の場合をも考慮する必要がある．

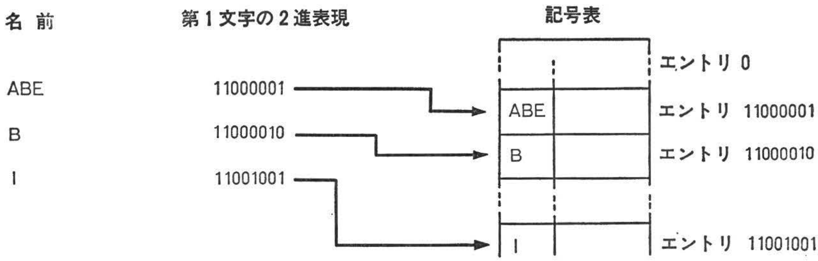


図9.2 ハッシュ番地

記号の第1番目の文字の内部表現というのは，簡単なハッシュ関数の一つである．例えば，A の 2 進表現が 11000001 であれば，記号 ABE をハッシュすると 11000001 (16 進表現で C1，となる．見出し ABE を探す時に最初に見るエントリが第 11000001 番目のそれである．図9.2 は，名前 ABE, B, I のエントリを示している．

二つの記号をハッシュして同じ番号にならないかぎり，表引きにかかる費用はハッシュにかかる費用そのものである．したがって，n 個のエントリをもつ，整列されていない表を引く場合（比較回数が平均 $n/2$ 回），驚くほど時間を節約できることになる．

しかしながら，2個の記号をハッシュして，同じ番号になったときには，面倒が生ずる．これを**衝突** (collision) と呼んでいる．その番号のエントリに置くことができる記号は，当然1個だけだから，他の記号のために別の場所を探してやらねばならない．衝突がたびたびおこらないように，ハッシュされた番号が表の全域に，一様に拡がるようにするのが良いハッシュ関数である．上例のようなハッシュ関数は1文字目が同じ名前を，全て，同じ番号にハッシュしてしまうので，悪いものの見本である．ここで，別のハッシュ関数をいくつかみてみよう．衝突の問題を解決するための方法として，**再ハッシュ法** (rehashing) と**くさり法** (chaining) について述べよう．

表のエントリは，全て，1語の長さとする．ハッシュ関数 h は，N 個のエントリをもつ表に対して，整数 $0, 1, \cdots, N-1$ をつくりだす．もし，1 エントリが，k 語からできていれば，これに k を乗じて表の先頭番地に加えてやればよい．

9.3.1 再ハッシュ法 (Rehashing)

もし記号 S をハッシュして，その番号 h のエントリがすでに別の記号に占有されていたとしよう．すなわち衝突である．このとき，p_1 を適当な整数とし，エントリ $h+p_1 \pmod N$ の内容と S とを比較する (N はテーブルの大きさ)．再び衝突が生ずれば，適当な整数 p_2 に対して，エントリ $h+p_2 \pmod N$ の内容と比較する．これを $h+p_i \pmod N$ 番目のエントリが空であるか，S を含むか，再び h となってしまう ($p_i=0$) までつづける．最後の場合は，テーブルが一ぱいになっているので，プログラムを停止しなければならない．

i 回の衝突が生ずると，$i+1$ 回目の比較をエントリ $h_i=h+p_i \pmod N$ と行う．p_i の値は，比較回数の期待値 E が小さく，できるだけ多くのエントリをおおうように選んでやらなければならない．$0, 1, \cdots, N-1$ を全ておおっていれば，理想的である．p_i を求める4通りの方法を述べよう．

再ハッシュ法は，**分散記憶** (scatter storage) ということばと結びついているのが普通である．それは，データのはいったエントリが，表中にばらまかれているからである．注意すべきは，表のエントリは，実行に先立って，記号になりえないような適当な値で全て初期設定しておかねばならないことである．これによって，すでに使用されているエントリと空のエントリを区別することができる．さらに，表の全エントリの場所を最初から用意しておかねばならない．また最初に用意した表が一ぱいになったからといって，表を拡げようとして表中の全ての記号のハッシュ番号を計算し直し，それらを新しい番号に対応するエントリに移し直すことは容易なことではない．

線形再ハッシュ法 (Linear Rehash)

昔から知られている再ハッシュの方法 (おそらくもっとも能率の悪いもの) は，$p_1=1, p_2=2, p_3=3, \cdots$ とする方法である．このようにして，次のエントリと比較する．例えば，記号 S1 と S2 がハッシュされて，それぞれ2番と4番のエントリに入ったとしよう (図9.3 **a**)．ここで記号 S3 をハッシュすると2になったとする．衝突が起ったので，エントリ3に入ることになる (図9.3 **b**)．最後に記号 S4 のハッシュ番号が，再び2になるとすると，S1, S3, S2 の順で3度衝突がおこり，やっと図9.3 **c** の5番目のエントリに落着くことになる．

この方法の効率の悪さは，この例で明白であろう．このような方法で何回か

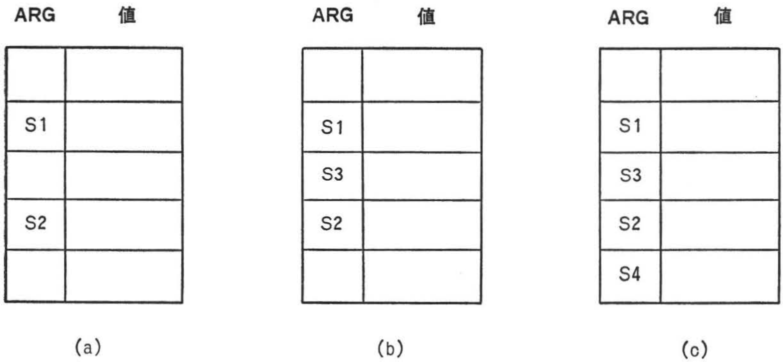


図9.3 線形再ハッシュ法

衝突を解決していると，エントリがいくつかづつかたまって，値の入ったエントリの長いくさり（団子）ができてしまう．空のエントリを探すために要する比較回数の期待値は，近似的に

$$E = (1 - lf/2) / (1 - lf)$$

で与えられる．ここで lf は，表占有率である．したがって，表占有率が 10 %であれば，1.06 回の衝突がおこりうる．もし 50% であれば，1.5 回，90 %であれば，5.5 回の衝突がおこりうることになる．期待値 E は，表の大きさには関係なくて，占有率だけに依存していることを注意しておく．

$p_1, p_2, \cdots$ としてもっと適当な値を見出すことができれば，この方法は，二分探索法よりずっと速い方法である．1024 個のエントリを持つ表で占有率が 50 % の場合を考えてみよう．512 個が使用中なので，二分探索法では，9 ～10 回の比較が必要なのに対して，この方法では，平均わずか 1.5 回で済んでいる．なお，整列済みの表や未整列の表の探索時間は，表の大きさではなく，現在のエントリ数に依っている．

ランダム再ハッシュ法 (Random Rehash)

この方法は，線形再ハッシュ法において，エントリが団子につながることをさけるために，整数 p_i を，疑似乱数として与える方法である．つぎの方法は，表の大きさ N が 2 のベキ ($N = 2^k$, k は正整数) のときに，具合よく行

く (Morris 参照)：

1. 処理ルーチンが呼ばれると，R を 1 に初期設定する．
2. p_i はつぎのようにして求める．
   a) $R=R*5$;
   b) R の下位 $k+2$ ビットを取出して改めて R の値とする．
   c) R の値を取出して，右に2ビット桁移動したものを p_i とする．

エントリの団子化を妨げるためのこの方法の重要な性質は，$p_{i+k}-p_i$ の値がすべて異なっていることである．衝突回数の期待値の近似式は

$$E=-(1/lf)\log(1-lf)$$

がよく合う．ここで lf は表占有率である．したがって，占有率が 10 % であれば，衝突は平均 1.05 回，50 % であれば，1.39 回，90 % になれば，2.56 回となる．

加算型再ハッシュ法 (Add-the-Hash Rehash)

h を最初のハッシュ番号として，$p_i=i\times h$ とする．すなわち，エントリ $h, 2h, 3h, 4h, \cdots$,（これらは全て，表の大きさ N を法として計算した値とする）で比較を行うことになる．N が素数であれば，上の系列は，表の全ての範囲を，平等におおうことなるので，この方法は具合がよい．衝突が起きた時につぎのハッシュ番号の計算は非常に簡単である．しかしながら，最初の番号 h が 0 になってはまずいので，h は $0, 1, 2, 3, \cdots, N-2$ ではなくて，$1, 2, \cdots, N-1$ のいずれかの値をとらなければならない．

2次再ハッシュ法 (Quadratic Rehashing)

この方法は $p_i=ai^2+bi+c$ を用いる．ここで，a, b, c にはどんな値が用いられてもよい．しかしながら，重要な点は，表中の多数のエントリが p_i によっておおわれることが確かでなければならないことである．例えば，$p_1=5$, $p_2=10$, $p_3=5$, $p_4=10$, $p_5=5$, $p_6=10$ であれば，h, $h+5$, $h+10$ の高々3個のエントリでしか，比較が行えないのである．

表の大きさが2のべきであれば，エントリの数は，非常に少なくなってしまうことがわかる．表の大きさが，素数であれば，この方法で，表のちょうど半分をおおうことができる．もちろん，この方法は，ランダム再ハッシュ法ほど

はよくないが，平均衝突回数と p_i を計算するのに要する時間は，少くなっている．

係数 a, b, c の値は，計算機に依存してよいので，計算機によっては，機械命令に適した a, b, c の値を選んで，能率のよい計算を行うことができる．IBM 360 を例にとって，能率のよいプログラムを作る係数の選び方を説明しよう．先頭番地を HTBL とし，787 個のエントリをもつ表を仮定する．

$c = 0$ とすると，最初のエントリは $h + p_0$ となり，$p_0 = 0$ である．$p_{i+1} - p_i = 2ai + a + b$ であるから，この式で，i の係数を -1 にするために，$a = -1/2$ とすると

$$p_{i+1} - p_i = -i - 1/2 + b$$

が得られる．したがって，i が 1 上がると式の値は 1 下がる．レジスタ INC の初期値を $-1/2 + b$ にして始めると，次の値の計算は，レジスタから 1 を引くだけでよい．さらに，表をちょうど半分探索したところで，計算を止めるのにレジスタ INC を使うこともできる．もし $b = 392 + 1/2$ にしておけば，393 回の比較を行って，INC $= -1$ になったときに，計算を停止する．p_i を求める式は

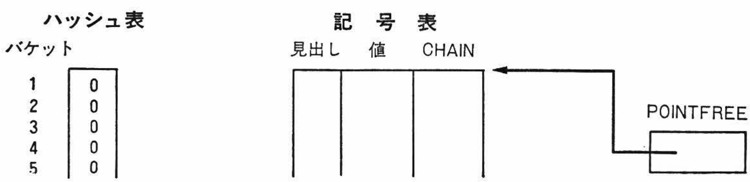
$$p_i = -(1/2)i^2 + (392 + 1/2)i$$

となる．IBM 360 では，表の要素は，4 バイト単位で，普通の番地計算は，いつも 4 倍して行うことに注意しよう．以下に示す IBM 360 のアセンブリ言語によるプログラムでは，これを用いている．再ハッシュを明らかに示すために，挿入すべきエントリとして 1 ではなく，0 に等しいものを探すプログラムとなっている．

```
*       Registers used.              使用されるレジスタ.
HASH    EQU   2                      現在のハッシュ番号. 初期値 h.
FR      EQU   3                      −4 が入っている (4 バイト/1 エントリ)
INC     EQU   4                      p_{i+1}−p_i の値をいれる.
KN      EQU   5                      表のバイト数の総計 :787×4
ZO      EQU   6                      0 が入っている.
INIT    L     HASH, H                最初のハッシュ番号 (0, 4, 8,…) をとり出す.
        LA    INC, 1568              最初の増分 (b−1/2)*4.
        LA    KN, 3148               表の大きさ=787*4.
        SR    ZO, ZO                 ZO を 0 にする (比較回数).
        B     COMP                   第 1 回の比較へ.
COLIS   BXLE  INC, FR, FULL          INC の値を −4 だけ減らす. 結果が −4 なら,
*                                    衝突回数が多すぎたので FULL へ (この名札
*                                    はプログラム中には定義されていない).
        BXH   HASH, INC, COMP        ハッシュ番号に INC を加えて h+p_i を得る.
*                                    h+p_i<KN (レジスタ 5) ならば, つぎの命令
*                                    をスキップして COMP へ.
        SR    HASH, KN               h+p_i mod 3148 (表の大きさ). エントリは空か,
COMP    C     ZO, HTBL(HASH)         衝突が起れば, COLIS へ行き INC を求めて
        BNE   COLIS                  次のエントリを調べる.
*ここにくるのは, HTBL(HASH) の内容が空の場合.
```

9.3.2 くさり法 (Chaining)

連鎖構造を用いたハッシュ技法は，ハッシュ表と記号表とポインタ POINTFREE とを用いる．ここでハッシュ表とは，バケットと呼ばれる，初期値 0 の要素からなる表であり，記号表も初期値 0，POINTFREE は，記号表の現時点の最後のエントリを指すポインタで，最初は表の先頭より一つ前の番地を指している．記号表のエントリには，見出しと値のほかに，CHAIN という欄が加わりこれにはゼロまたは他のエントリの番地を入れるようになっている．したがって，初期状態は下のようになる．

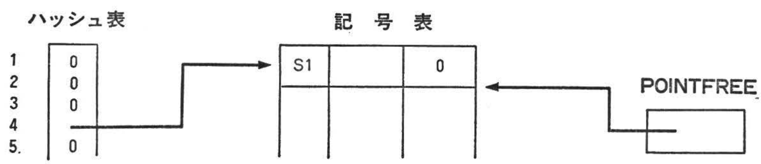


記号は，ハッシュ表のバケットの番号にハッシュされる．バケットの中は，ゼロか，あるいは求める記号を最初にハッシュして得られた番号に対応する記

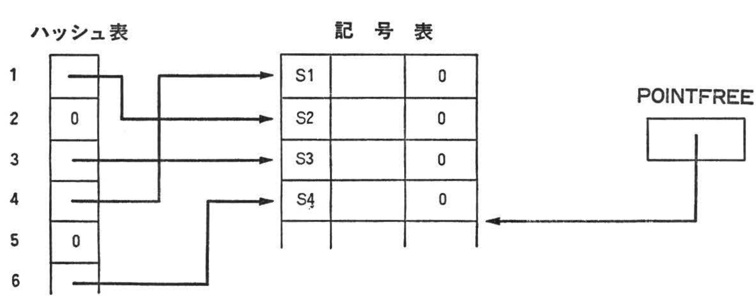
号表のエントリを指す番地が入っている．各エントリの CHAIN 欄は，ハッシュして同じバケット番号になった記号のエントリをくさりにしてつなぐために用いられる．この処理過程を丁寧に見てみよう．まず上のように，記号表が空である．ここに記号（例えば）S1 をのせるとしよう．ハッシュ関数が，ハッシュ表の番地を生成する．4番のバケットとしよう．この中は，この時点ではゼロである．ここで次のようにする．

1. POINTFREE に 1 を加える．
2. (S1，値，0) というエントリを記号表中の POINTFREE で指されるエントリに挿入する．
3. POINTFREE の値をバケット 4 にいれる．

この手順を踏むとつぎのようになる．

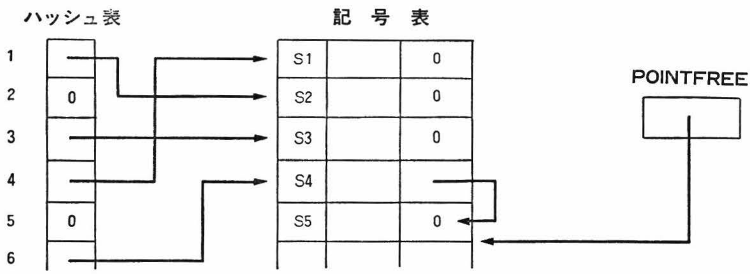


記号をハッシュして，それぞれ別のバケットに入ってゆくかぎり上のように挿入をつづければよい．もし記号 S2，S3，S4 をハッシュして，それぞれバケット 1，3，6 に入れば，二つの表は

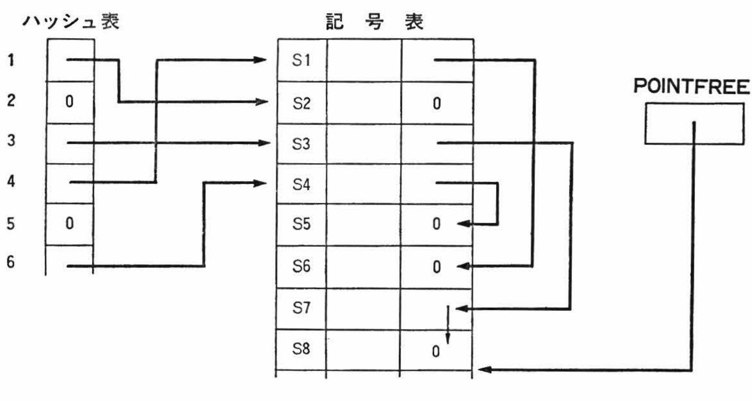


のようになるだろう．しかしたまたま記号 S5 をハッシュするとすでにふさがっているバケットに入ることになるとしよう．ここで CHAIN 欄の活躍が

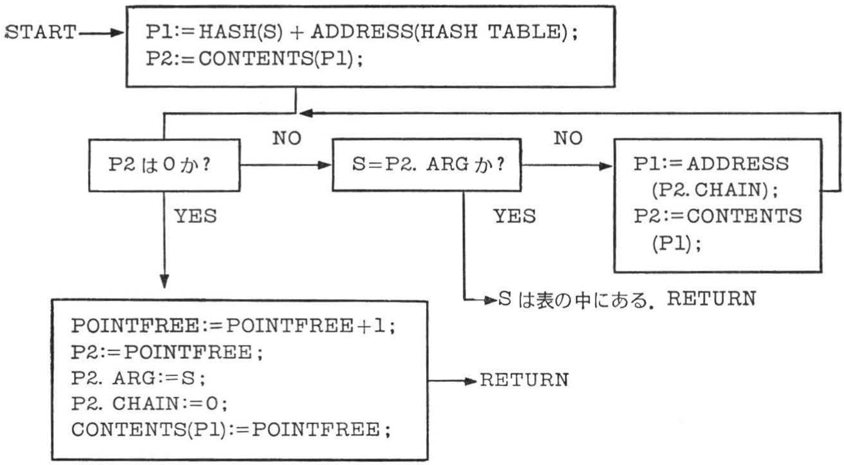
始まる．S5 を記号表におさめ，そこにそのバケットのくさりのはしを加える．もし S5 のハッシュ値がバケット 6 であれば，つぎのような構造が得られる．



それぞれバケット 4，3，3 になるような記号 S6，S7，S8 をこの順に表に挿入するとつぎのような構造が最終的に得られる．



ここで，記号 S が与えられたときに，表を探索して，もし表中に S がなければ，それを表にのせる手続きの流れ図をつぎのように与えることができる．この流れ図で，P1 と P2 は一時的なポインタ変数，HASH はハッシュ関数で，その引数，記号 S，からハッシュ表中のバケットの番地を計算する．手続きを終ると P2 には，エントリの番地が得られている．



もし，記号表が一ぱいになると，記号表に別の一連のエントリをもってきて追加することがつねに可能である（ただし，言語あるいはシステムが動的記憶割付を許すとしてのことであるが）．なんとなれば，ハッシュ関数はハッシュ表の指標を与えるだけであり，記号表のエントリは，ポインタを通じて参照されるにすぎないからである．したがって，再ハッシュ法のように，エントリの最大数というものはない（しかしながら，バケットの最大数はある）．さらに，バケットだけは初期設定しなければならないが，記号表のエントリはその必要がないことにも注意されたい．バケットの数はしばしば 100 ～ 300 位であるが，実際の記号表のエントリの数はそれよりずっと多いのである．一たび全てのエントリを登録してしまうと，ハッシュ表を投げだして，その記憶場所は他の目的のために解放することができる（このとき全ての名前は，記号表のエントリ番号あるいはそれと同等なもので置換えられるものとする）．各エントリにはくさりの場所のために1語余分に必要とするが，これは，また種々の場合に役に立つ．

9.3.3 ハッシュ関数

ハッシュすべき記号 S が，計算機の1語の長さを超えるならば，ハッシュ関数の第 1 段は，S から，計算機の1語におさまる S' を作り出すことであ

る．（例えば，IBM 360 では，1 語は 4 文字にあたる.）S' は，S の全ての語を加えたり全ての語の排他的論理和（EXOR：1 EXOR 1＝0, 0 EXOR 0＝0, 1 EXOR 0＝1, 0 EXOR 1＝1）をとったりして作るのが普通である．

第2段は S' から最終の指標を計算することである．これには，いくつかの方法がある．

1. S' に S' を乗じて，まん中の n ビットをハッシュ値とする（ここで，表は 2^n 個のエントリを持つものとする）．まん中の n ビットは，S' の全てのビットに依存するので，これは良い方法である．
2. S' の適当な部分に，排他的論理和のような適当な論理演算を施す．
3. 表が 2^n 個のエントリをもつならば，S' を n ビットずつの部分にわけて，それらを加え合せ，結果の下 n ビットを用いる．
4. S' をテーブル長で割り，剰余をハッシュ値に用いる．

これらの方法は全て使われて満足すべき結果を得てきている．このほかにもいろいろ工夫できるだろう．しかし，使われる可能性のある記号の種類に対して十分ランダムになるような番地を生成するようなハッシュ関数を選ばなければならないことだけは注意しておかねばならない．われわれの場合，記号表には最初からその言語の予約語と標準関数の名前が設定されているとすれば，最初の表の中で，実際どのくらい衝突が生じているか，検査してみると良い．統計的にみて，十分良いハッシュ関数を作ることができても，予約語のハッシュのうち 10 個が同じ番地になってしまうことがあるかもしれない．

ここで全ての英大文字と数字の EBCDIC コードの左側の2ビットが (11) であり，同じく英大文字の ASCII コードの左側の 2 ビットが (10) であることに言及しておかねばならない．ハッシュ関数は，この位慎重に作らねばならない．

PL/I（水準 F）コンパイラ（IBM (b) 参照）はつぎのようなハッシュ関数を用いている．

1. 名前を連続した4文字づつに区切って全て4バイトレジスタに加えてゆく．
2. 結果を 211 で割り，余りを R に得る．
3. 211 個のバケットを持つハッシュ表の $2*R$ 番目を指標とする（バケットは2バイト長）．

原始プログラム中の名前と定数は全て，記号表のエントリの参照に置換えてしまう．これ以上エントリを作る必要がなくなると，ハッシュ表を放棄する．エントリは可変長である（最小4バイト＋EBCDIC 名）．

9.4 木構造を持った記号表

この方法はエントリの順序付けに2進の木を用いる．2進の木とは，各枝に1個または2個の節点を持つ木である．木の各節点は値の入った表のエントリに対応する．根にあたる節点はエントリ1である．図9.4**a** は名前 G だけのエントリをもった表を示している．ここで名前 D が入ってきたとしよう．D＜G であるから，枝を左側に描いて，図9.4**b** のようになる．名前 M がつぎにくるとしよう．G＜M であるから，枝は G から右側に描く（図9.4**c**），つぎに名前 E がくるとしよう．E＜G であるから，左の枝をたどって D に行当る．ここで D＜E なので，D から右側に枝を描く（図9.4**d**）．図9.4**e** は，このあと名前 A, B, F の順に追加された後の木を示している．

上の過程は，各エントリにポインタ欄を2個設けることによって実現することができる．一つは左側の枝を指し，残りは右側の枝を示す．必要な比較回数は，名前が現れた順序に大いに依存する．例えば，もし，名前が，A，B，C，D，E，F の順にやってきたとすれば，F を見出すには，6回の比較が必要である．最大探索回数を最小にするために，一たんでき上った木から，バランスのとれた木を再構成する段階が別に必要である．ここに述べたような2進の木を用いると名前の一覧表を辞書式順序で印刷するのが，非常に容易にできる（演習問題 1 参照）．

FORTRAN H コンパイラ（IBM (a) 参照）は，このような木を6本持っ

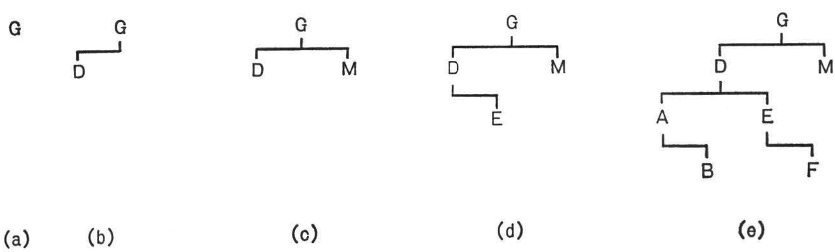


図9.4　2進の木による構成

ており，それぞれは，1，2，3，4，5，6 文字の名前のために使われる．4 バイト，8 バイト，16 バイト長の 定数に対しても別々の木を用意している．文の番号にも1本の木をあて，そのほかにもいくつか用いている．全てのエントリは同一の表に入っていて， 各 エントリは 52 バイト（13 語）からなっている．

演習問題　9.4

1. 2進の木の節点にある名前を，アルファベット順に印刷するアルゴリズムを作成せよ．**ヒント**：現在，木の上でどの点にありそこにいかにして行くかを示すためにスタックが必要である．

9.5　ブロック構造を持った記号表

ALGOL 型 の 言 語 は，ブロックや手続きが入れ子構造になっている．同じ名前を何度も別々のブロックや 手 続 き で宣言して使うことが許されているので，そのような宣言は，記号表中にそれぞれ別々のエントリを持たなければならない．ある名前を使用する場合，その名前のエントリを記号表の中から正しく発見することが問題となる．

原始プログラム中のブロック番号をブロックの開始した順に付けることにする（これは **begin** があらわれる順序と同じである）．これは左から右に構文解析を行えば，出会う順であるから，自然の順序付けである．

```
BEGIN REAL a,b,c,d;
        :
      BEGIN REAL e,f;
         L1:
      END;
      BEGIN REAL g,h;
               :
         L2: BEGIN REAL a;
                      :
               END;
          L3:
      END;
END;
```

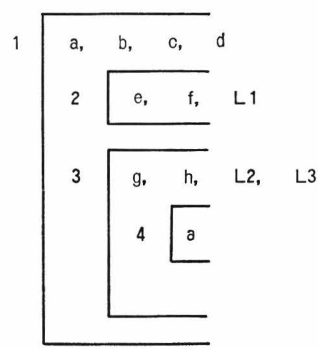


図9.5　ブロック構造

使用されている名前に対応する正しい宣言を見出す規則は，第一に，現在のブロック（名前が使用されているブロック）を探すこと，つぎに，それを直接囲むブロックを探すというようにつづけて，その名前の宣言を見付ける．このような探索法は，記号表のエントリを全て，各ブロックごとに，一つづきにしておくことと，**ブロック一覧表** (block list) を用いることとによって実現することができる．しばらく，各ブロックごとのエントリが順序付けられていないとしておこう．ブロック一覧表のエントリ i は，3 個の欄からなっていて，すなわち，ブロック i を直接囲むブロックの番号 (SURNO)，ブロック i に対する記号表のエントリの個数 (NOENT)，これらのエントリへのポインタ (POINT) である．ブロック一覧表のエントリは，3-組 (SURNO, NOENT, POINT) である．

図9.6 は 図9.5 のプログラムに対する記号表の最終版を示している．このような“ブロック構造を持った”記号表は，何らかのブロックが入れ子になっていてブロックの中で宣言された記号は，そのブロックの内側でしか参照できないという時にはいつでも使用できる．例えば，for ループはブロックと同じように入れ子になっているので，各 for ループごとに内部的にブロックを作り出してやる．こうすれば，for ループの外側からの飛込みを自動的に検出できるだろう．なんとなれば，名札の参照が可能なのは for ループのブロックの内側からだけであるとすれば，for ループの内側の名札は今述べた表引きの手続きによってしか見出すことができないからである．

手続きや関数の宣言の際に，仮引数や手続きの本体の外側に特別のブロックを作り出すこともできる．こうすれば，手続きの外から仮引数を参照すること

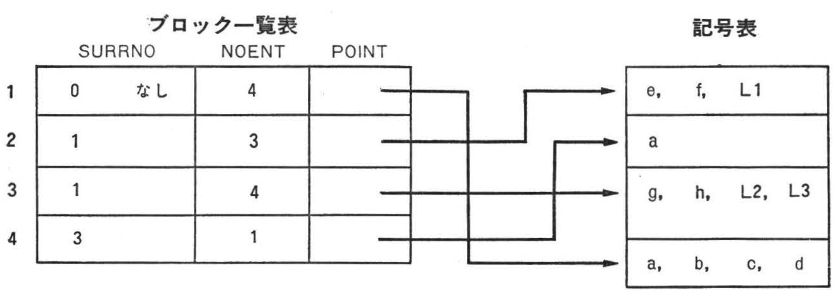


図9.6 ブロック一覧表と記号表

を防ぐことができる（手続き名自体は，もちろん，この内部で作ったブロックに属してはいない）．

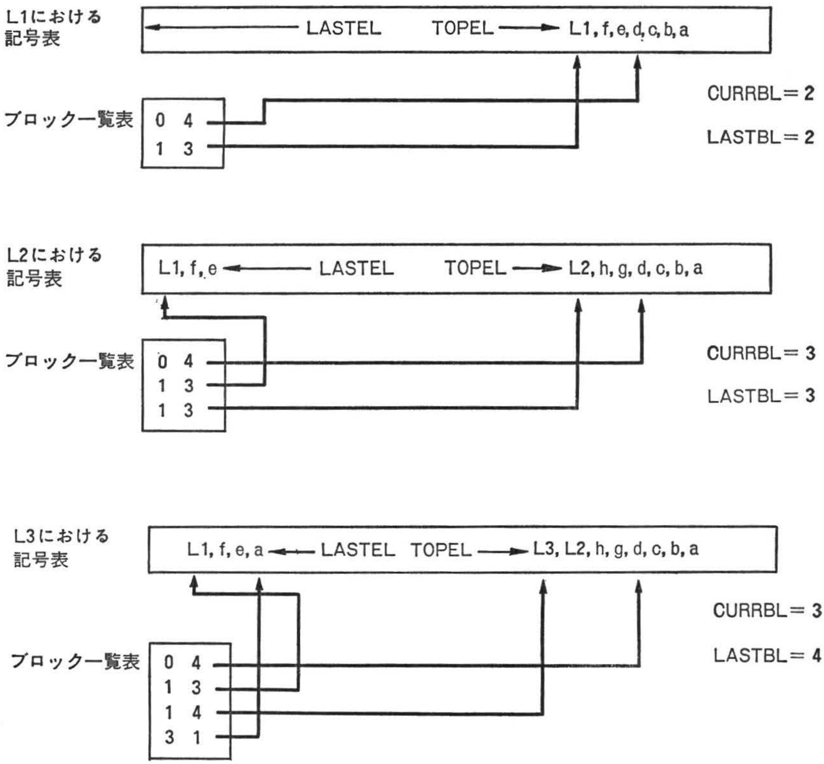
ブロックの開設と閉鎖

図9.6の記号表中のブロックは，2, 4, 3, 1 の順になっている．この順序は，実は，ブロックの閉鎖の順序（すなわちプログラムの中で，そのブロックに対応する **end** の現れた順序）になっている．これは各ブロックに対するエントリがそれぞれひと続きになっていなければならないという事実の当然の結果である．表を組立てる過程を実現するために，記号表の終りの部分（数語）をスタックとして用いる．このスタック S(N), S(N－1), … は，各ブロックの **begin** から始めて，構文解析を進めて，対応する **end** に解析が進む直前までのエントリを全て入れておく．**end** まで処理が終ってブロックが完成すると，このブロックに属するエントリは，表の前の方に移動する．もしブロックの構文を

(1) 〈ブロックの頭〉 ::= BEGIN
(2) 〈ブロック〉 ::= 〈ブロックの頭〉〈宣言部〉;〈文の並び〉END

と考えると，規則(1)の意味付けルーチンが実行されるきにブロックが“開設”され，規則(2)の意味付けルーチンが実行されるときにブロックは“閉鎖”される．図9.5のプログラムの構文解析において，名札 L1, L2, L3 の各時点における表の構成を以下に示す．ここで必要となる変数はつぎのとおり：

S(1:N)	N個のエントリからなる記号表 S(N), S(N－1), … はスタックとして使用．全てのエントリは最後は，S(1), S(2), … の順になる．
B(1:M)	ブロック一覧表．各エントリは SURNO, NOENT および POINT の整数型の3-組からなる．
INTEGER CURRBL	現在のブロック番号（初期値は 0）．
INTEGER LASTBL	割当てられたブロック番号の最大値（初期値は 0）．
INTEGER TOPEL	スタックの最上段の番号（初期値 N＋1）．
INTEGER LASTEL	記号表の使用されているエントリの最後の番号（初期値は 0）．



ここでブロックの開設と閉鎖の手続きをつぎのように与えることができる.

1. ブロックの開設. ブロック一覧表にエントリを1個設け, 現在のブロックの番号をこの番号とする: LASTBL := LASTBL + 1; B(LASTBL) := (CURRBL, 0, TOPEL); CURRBL := LASTBL.

2. ブロックの閉鎖. ブロックの内部で宣言された変数を記号表の正規の位置へ移す: B(CURRBL). POINT := LASTEL + 1;

```
FOR i := 1 STEP 1 UNTIL B(CURRBL).NOENT DO
BEGIN LASTEL := LASTEL + 1; S(LASTEL) := S(TOPEL);
      TOPEL := TOPEL + 1
END.
```

ブロックを直接囲むブロックの番号を調整する：

```
CURRBL := B(CURRBL).SURNO.
```

名前の登録と表引き

現在のブロックに新しく名前を登録するということは，記号表のスタックの最上段のブロックにその名前をのせて，そのブロックの名前の個数を1あげることである．ここでは，探索の手続きが2個必要である．一つは，現在のブロックの中だけの探索であり（例えば，宣言部），もう一つは，現在のブロックとそれを囲む任意のブロックを探索するものである．手続きを書くのは簡単であり，読者に任せよう．

議　　論

宣言が行われる前に変数を使うことができる言語では，記号表が完全にでき上るまでは，変数の参照を記号表のエントリと関連させることができない．そこで，記号表を作り上げるという目的をもったパスをまず行わねばならない．これは，言語に対して，非常に簡単な文法を使用するだけで，大部分の文は，そのまま素通りさせてしまう．このパスが見なければならないのは，宣言文，名札の定義，ブロックや手続きの構造，そして，おそらく for ループになるであろう．次のパスで，すでに作成されている記号表を用いて，プログラムの完全な解析ができる．

宣言を行う以前に変数を用いることを許していない言語もある．この制限は，コンパイルの手続きを容易にするためだけにおかれているのが普通である．しかしながら，名札は，その定義よりも前に参照されることがよくあるので，定義する前に，あらかじめ名札を宣言しておくことをプログラマに求めることは，奨められない．第13章では，このような名札を一回のパスで処理する方法を論じている．

実際の翻訳が始まる前に全ての原始プログラムを囲む疑似的なブロックが設定されて，そこには，標準関数や標準的な変数のエントリが設けられる．これらの名前を別に取扱う必要がないからである．

1パスのコンパイラでは，ブロックが閉鎖されるとそれに属する名前は，もはや必要でない．そこで，そのエントリをスタックから，記号表の正規の位置に移すかわりに，そこから削除してしまうだけでよい．そのような表を**スタック記号表**と呼ぶ．

ハッシュ技法とブロック構造

ハッシュ技法を使うと記号表の中をブロックごとに並べるのが容易でなくなる．ブロック構造を取扱うのにまず頭に浮ぶ方法は，ブロックの番号と名前とをつなげてハッシュ関数の引数にしてやる方法である．こうすれば，どんな名前も一意的となる，名前 B を探すには，まず現在のブロック番号を B の頭につけて行い，つぎはそれを直接囲むブロックの番号をつけて，探索し，この手続きを，名前が見付かるまでつづければよい．

一方，各エントリにもう一つポインタ欄を設けて，同じ名前で別のブロックにあるもののエントリをこれとくさりでつなぐ方法がある．一たんくさりのはしが見つかれば，くさりをたぐって正しいエントリを探せばよい．不幸なことに，新しいブロックが開始するまでに，あるブロックの宣言を，完全に走査し終ってないことがしばしばある（これは，例えば，手続きの宣言などで起る）．したがって，例えばブロック 3 の名前 B のエントリがブロック 1 の名前 B のエントリの前に来てしまうこともありうる．そこで，くさりの上のエントリがどんな順序で現れるのか確かめることができなくなる．表引きを能率よく行うためには，エントリをくさりにつなぐときや，ブロックを閉鎖するときに，くさり上のエントリを並べなおす必要があろう．

9.6 参考文献概説

整列や表引きのアルゴリズムについては，Flores (69) や Kunth (70) を参照してほしい．プログラミング入門書にも，簡単な整列法の章が大抵設けられている．ハッシュ技法も，長年にわたって多くのプログラマによって，開発されてきたプログラミング技法の一つである．Peterson (57) の論文が最初のものである．この論文で，彼は線形再ハッシュ法を論じている．ランダム再ハッシュ法（とその解析）は，Morris (68) の概説論文にある．これは，Vyssotsky の方法の変形である (McIlroy (63) を参照)． 2 次再ハッシュ法は Maurer (68) による．最近，これは，二つの方向に拡張されている．Radke (70) は，この方法によって探索されない残りの半分を探す問題を解決した．CACM の同じ号で，Bell (70) は，$p_i = ai^2 + bi$ のかわりに，$p_i = a(h)i^2 + bi$ を用いることによって，この方法を拡張した．すなわち，i^2 の係数が，最初のハッシュ番地 h の関数になっている．彼は，この方法で，衝突回数の期待値が小さくなることを解析的に示し，実験例を与えている．

Batson (65) は ALGOL コンパイラで線形再ハッシュ法を用いたことを報

告している.

Morris (68) も衝突の問題を解決するためにくさりでつなぐ方法の解析を与えている. この解析は, Johnson (61) のものが最初である.

2進の木に関するより詳しい議論は Kunth (68) の第1巻にある. ここで述べたブロック構造の記号表を構成する2通りの方法は, 多くのコンパイラで用いられている. したがって, これらが, 特に, 誰が考えたものときめることは大変難かしい.

第10章

記号表の中のデータ

10.1 記述子

名前に関する情報は全て記号表の"値"の欄に格納されている (9.1節参照). 原始プログラムの中で名前が一つ現れるたびに, 記号表の中にそのありかを探し, そこにすでに入っている情報とその名前が現れた場所で得られるその名前に関する情報とを比べて, なにか新しい情報があればそのエントリに追加する. このように記号表はコンパイラの中で非常に重要な役割を持っている. ある意味で全ての翻訳の過程はこの表のまわりをぐるぐる廻っているといってよい. この表は, これらの情報を能率よく取出し, 能率よくおさめることができるような構造になっていることが望ましい. (同時に, 名前の数が多い大きなプログラムも収容できるように表の各要素はできるだけ小さいことが望ましい. いわゆる時間対空間の問題である.) 一般に, 記号表に直接手を下せるのはできるだけ少数のルーチンに限っておけば, 変更が必要なときは, 容易に行える. どんなプログラムでもそれを作る前に記号表の値の欄の形式をできるかぎり完全に決めておくのがよい.

記号表の"値"欄は, 名前に関する記述をしているので, これに**記述子** (descriptor) という術語をあてよう. Feldman (66) は, これに対して, **意味語** (semantic word) という言葉をあてている.

記述子に必要な情報の量は, 名前が実際にどんなものか——単純変数か配列か関数かなど——に依存する. したがって, 記述子が可変長のコンパイラもある. これは表づくりの全ての場合に可能というわけではない. 記憶場所の無駄遣いをしないためには, 記号表の1単位の要素の大きさは小さくし, 1単位を超える記憶場所を必要とする名前に対しては, 表の要素を2〜3個続けて割当

てる．このほか，必要とあれば記述子の一部をポインタ用にとって2次的な表を設けこれを指すようにする方法もある．この場合，エントリの種類によって，記述子欄の内容の意味が変ってくる可能性があるので，注意深くプログラムする必要がある．

変数や手続きに対しては，記述子の中につぎのような情報が必要である．

変　数：型（実数，整数，文字，複素数，名札など）;
　　精度，倍率，長さ;
　　形式（単純変数，配列，構造体など）;
　　実行時の番地;
　　配列ならば，その寸法．寸法が定数であればその値;
　　構造体かその構成要素ならば，エントリは，何らかの方法で，他の構成要素と連係していなければならない;
　　仮引数かいなか，もし仮引数なら，引数の結合方法の種類;
　　(FORTRAN の場合) COMMON 文あるいは EQUIVALENCE 文の中に現れたか．もしそうならば，関係する名前と結びつけなければならない;
　　この名前は宣言ずみか;
　　この名前に値を代入するような文があるか;
手続き：このプログラムの外部で宣言されたものか;
　　関数か，その場合，型は;
　　再帰的か; 宣言ずみか;
　　仮引数は何か，仮引数は実引数と対応させるために，手続き名と結びつけておかなければならない．

記述子の中に入っているものを示すには，現存のコンパイラの例を与えるのがおそらく最良の方法だろう．別々の機械の上の二つの言語の例をここでお目に掛けよう．ただ，これらのコンパイラの説明文書で用いている記号や形式を多少変え，ここで不必要な情報は省いていることを断わっておこう．つまり，ここでは細大もらさぬ記述を与えるわけではない．

ALCOR Illinois 7090 コンパイラ [Gries, Paul, Wiehle]

これは，IBM 7090 に作った4-パスの ALGOL コンパイラである．第1パスは，宣言，名札，ブロック，手続きそれに for ループを見て，ブロック構造

を持った，未整列の記号表（ブロック一覧表付き）を作りあげる．この時点では，属性は一つも書込まない．第2パスでは，推移行列法を用いて，プログラムの構文解析を行い，それぞれの名前ごとにその記述子に書込んで行く．実行時の記憶場所を割付け，for ループの最適化のための前処理を行う．第3パスでは，命令を作り出し，第4パスでは，メッセージを出力し，プログラムを，ローダにあうような形式に整える．

IBM 7090 は1語36ビットであり，15ビットの番地部を持っている．そして 0-2, 3-17, 18-20, 21-35 ビットを別々に参照できるような命令がある．ALCOR Illinois 7090 コンパイラの記述子は，初め下図のように2語を占めている．2パス目を終えると，2番目の語は捨られて，その場所は，別の目的に使われる．

0−5	6−7	8	9−13	14−20	21−35
型 種類	仮引数	宣言済みか？	階層番号	添字または 引数の数	目的プログラム での番地

0−2	3−17	18−20	21−35
使用	エントリ番号		ループの検査のために 使われるポインタ

語	ビット	意　味
1	0-5	変数の型（2ビット）と種類（4ビット）を示す．例えば 000000 = これまで未定義 010001 = 整数型　単純変数 010010 = 整数型　配列 100001 = 実数型　単純変数 100100 = 実数型　関数 100101 = 実数型　定数 001111 = 　　　　名札
1	6-7	変数が仮引数であれば，ビット 6 = 1；さらに値呼びであれば，ビット 7 = 1.
1	8	初期値0．パス2で，宣言の処理を終ると1になる．
1	9-13	この宣言を行っている手続きの階層番号（入れ子の水準）．
1	14-20	配列名のとき添字の数，手続き名のとき引数の数，それ以外は0．
1	21-35	変数が宣言されている手続のデータ領域のオフセット．

2	0-2	名前呼びの仮引数に用いる．初期値は0．手続きの中でこの仮引数に値が代入されるならば1を置く．対応する実引数が定数ならば2．以上の二つは互に排反である．なんとなれば定数に値を代入することはないからである．大抵の場合，このような誤りを検出するのはこの方法による．
2	3-17	表におけるこのエントリのエントリ番号．
2	21-35	ループにおける最適化の可能性を調べるために使用．第18章参照．さらに，この欄は，誤りの回復のためにも使用する．第15章参照．

実行時の番地は，1対の値（データ領域番号，オフセット）で与えられることは第8章で述べた．上の表では，データ領域番号は使用しておらず，その代り階層番号（入れ子の水準）を用いている．なんとなれば，同じ階層番号を持った手続きは，並行なブロックにあり，したがって，コンパイラがそれらを同時に処理することは決してあり得ないからである．例えばBが，階層番号 i の手続きの中で宣言され階層番号 $k(i \leq k)$ の手続きの中で参照されているならば，k のデータ領域の i 番目の番地には，i のデータ領域の先頭番地が入っており，これが，B を参照するために必要なデータ領域であったことを思い出そう（8.9節参照）．このように，階層番号はわれわれの目的に適っている．

/360 WATFOR [Cress ほか (68)]

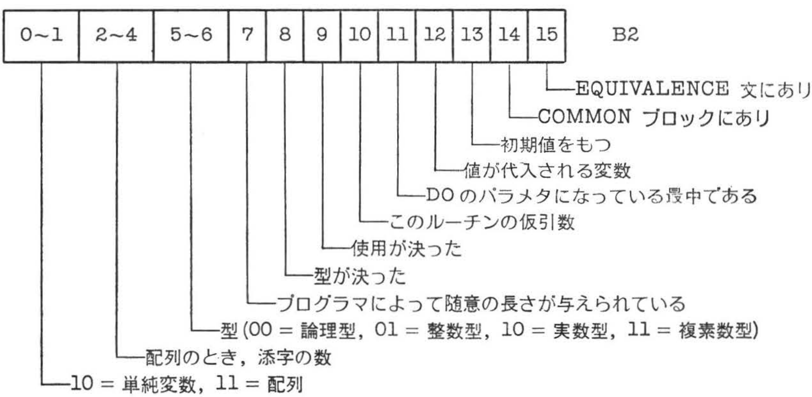
第1章で述べたように /360 WATFOR は，本質的には1-パスの FORTRAN IV コンパイラで IBM 360 で動く．

このコンパイラの記号表は，異なった5本の1次元リストからできている．すなわち文番号；数値定数；共通ブロックの名とサブルーチン名および変数名；共通ブロックの名およびサブルーチン名（したがって，これらの名は二つの表に現れる）．リスト上の全ての要素は，同一のテーブル上にあって，リスト構造で結合されている．すなわち，各エントリの最初の2バイトが，同じリストに属する次のエントリへのポインタに用いられている．したがって，特定のエントリの探索には1次元リストを頭から探す方法がとられる．エントリは種類が異なれば長さも異り，8～20バイトにわたっている．例えば，変数用のエントリは，下記のような16バイト長である．（ここで，最初の2バイトは省いている）：

バイト	3－4	5－10	11－12	13－14	15－16
	B2	名　前	寸 法	COMMON	EQUIVALENCE

COMMON 欄と EQUIVALENCE 欄はそれぞれ同じ COMMON ブロックに属する変数あるいは，EQUIVALENCE されているデータのエントリへのポインタに用いられている．これは14章で詳述する．寸法欄は，変数が添字付でなければ，0である．配列であれば，寸法の全ての情報（上限 $d_1, \cdots, d_n$ と配列の大きさ $d_1 * \cdots * d_n$）をおさめているエントリへのポインタである．

B2 と名付けた2バイトの欄は，下図に示すような情報に分割される．各欄とも初期値は0で，プログラムから情報が抽出されるにつれて，値がおかれる．1ビットの欄に，1が設定されているときの意味は下図に示すとおりである．



10.2 構造化されたデータの構成要素のための記述子

構造化データの構成要素のための記号表のエントリは，構造を完全に表現するために何らかの方法で，たがいに結合しておかねばならない．これには，いろいろの方法がある．とにかく，構造化データやその構成要素への参照を能率よく処理できるような表現がとられていることが望ましい．ここで取扱う構造は，以下の2例である：

```
(10.2.1)  1 A                  1 L
            2 B                  2 M
              3 C                  4 C
              3 D      および      4 D
            2 E                  2 B
            2 F                    5 C
                                   5 D
```

これらは COBOL や PL/I の構造体に似ている．左側の構造体は，A という名が付いており，三つの構成要素 B, E, F から成っている．E と F はこれ以上部分構成要素を含んでいないので**基本要素**† (elementary component) と呼ぶ．基本要素は型を持ち，値を代入することができる．ここでは，各構成要素同士の結合方法にのみ関心があるので型や代入については議論しない．B は基本要素ではなく，その（部分）構成要素 C と D の**集団**† (group) の名である．C と D は基本要素．構造体の名 A と L はまた集団の名でもある．

構造体を記述する規則は下記のとおり：

1. 名前の前にそれぞれ**水準番号** (level number) と呼ばれる正数を置く．
2. 構造体を記述する際に最初に記述する要素が構造体の名で，その水準番号は他の要素の水準番号より小さいものでなければならない．
3. 集団の名に続いてその部分構成要素を記述する．この部分構成要素は全て同じ水準番号を持たねばならず，これはその集団の水準番号より大きいものでなければならない．

ここでは，名前の一意性に関する規則は設けていない．

構造体は，全て木で表現できる．木の根にあたる節点が構造体の名に相当する．集団の名は，その集団の構成要素を節点にもつような枝の名前である．木の端の節点が基本要素にあたる．

集団の構成要素（それぞれ同じ水準番号を持っている）をしばしば**兄弟** (brothers) と呼ぶ．集団と名付けた要素は，**父** (father) であり，その構成要素を**息子** (sons) という．例えば，上の左の例で，A の息子は，B, E, F であり，これらは，互いに兄弟である．要素 B は，2人の息子 C, D を持ち，逆に C, D の父は B である．

ある構成要素を参照するには

$$A_n.\ \cdots.\ A_1.\ A_0$$

とする．ここで $n \geq 0$ であり $i=0, \cdots, n-1$ に対して，名前 A_i は集団の名 A_{i+1} の中に含まれた（必ずしも直接でなくてよい）構成要素の名前でなければならない．

---

† **訳注**　これらの用語は JIS COBOL のものとは異なる：**基本項目** (elementary item)，**集団項目** (group item)．

A_n, …, A_0 の系列が，A_0 を完全に“限定する”必要はないことを注意しよう．定義されている構造体が (10.2.1) の2個だけだとすると，A.B.C でも A.C でも構造体 A の集団 B に含まれる C を参照できる．一方，M.D でも L.M.D でも，構造体 L の集団 M に含まれる D を参照できる．しかし，L.C では，L.M.C とも L.B.C ともとれるので，あいまいである．B.C もあいまいである，なんとなれば，これは，A.B.C とも L.B.C ともとれるからである．

もし名前が単独で参照されるならば，それは，ただ一つに定まるものでなければならない．M とすれば，構造体 L の要素 M をただ一つ参照するが，B ではあいまいである．

A_n. … . A_1. A_0 を参照とするのに $A_0(A_1(\cdots(A_n)\cdots))$ という記法と A_0 OF A_1 OF … OF A_n という表記法とは，同等である．二つの表記法の差によって，構造体の表現法や処理法に実質的な差が生ずることはない．

記号表における構造体の表現形式

構造体を表現するには，要素一つにエントリ一つをあてる．エントリには普通の欄（これについては述べない）のほかに，FATHER, SON, BROTHER という三つの欄を付加える．FATHER 欄は，その構成要素の父にあたるエントリへのポインタが入る．SON 欄には，その要素の**最初の息子** (first son) へのポインタが入り，BROTHER 欄には，部分構成要素を1列に並べたときに，このエントリのつぎの弟のエントリへのポインタが入る．これらの欄に0が入っているとすれば，それは，そのエントリの父か，息子か，つぎの兄弟かがそれぞれいないことを意味している．**図 10.1** に (10.2.1) の二つの構造体のエントリを示した．この情報は，構造体を能率よく参照するには，多すぎるかもしれないし，少なすぎるかもしれないが，これは，これから明らかにして行こう．

同じ名前を持ったエントリがいくつか存在する可能性があるので，SAME という欄をもうけて，同じ名前をつなぐポインタを入れる場所にする．同じ名前を持ったエントリからなるリストの最初のエントリの SAME 欄には0が入る．このようにして，つぎのような5欄を持つエントリができあがる：

1. ID. 構成要素の名前;
2. SAME. 同じ名前を持った一つ手前のエントリへのポインタ（同じ名前

がなければ 0);

3. FATHER. この構成要素を直接含む集団の名前のエントリへのポインタ;

4. SON. この要素の最初の部分構成要素のエントリへのポインタ（したがってこのエントリは集団の名に対するものである．さもなければ，この欄は 0);

5. BROTHER. このエントリを含む集団に含まれる次の要素のエントリへのポインタ（なければ 0).

	ID	SAME	FATHER	SON	BROTHER
A1	A	0	0	B1	0
B1	B	0	A1	C1	E1
C1	C	0	B1	0	D1
D1	D	0	B1	0	0
E1	E	0	A1	0	F1
F1	F	0	A1	0	0
L1	L	0	0	M1	0
M1	M	0	L1	C2	B2
C2	C	C1	M1	0	D2
D2	D	D1	M1	0	0
B2	B	B1	L1	C3	0
C3	C	C2	B2	0	D3
D3	D	D2	B2	0	0

図 10.1　記号表における構造体の表現

記号表の組立て

図10.2 は（水準番号,名前）の対の系列で表される構造体に対する記号表を組立てるプログラムである．この対の値は，二つの配列 LEVEL と ID に順に入っており，*n* は，対の数を与える整数型変数とする．さらに，ADDENTRY(P, NAME) というルーチンの存在も仮定しよう．これは，NAME で示す名前に対する新しいエントリをとって，全ての欄に初期値0を与え，ID 欄と SAME 欄を埋め，エントリの番地をポインタPに返す目的を持っている．なお，各エントリには，各要素の水準番号を入れるために余分の欄 LEV を用いる．記号表が組上ってしまうと，この欄は不要となり，別の目的のために使用できる．この LEV 欄を使わないとプログラムが幾分複雑になる（演習問題1,2 参照）．

BEGIN POINTER P0, P, Q;	P0は,構造体の名前のエントリへのポインタ，Qは，現在処理中の構成要素のエントリへのポインタ，PはQの弟または父へのポインタ．
ADDENTRY(P0, ID[1]); P:=P0;	第1要素をエントリへ入れる．
P0. LEV:=LEVEL[1];	
FOR i:=2 STEP 1 UNTIL n DO	構成要素をたどるループ．
BEGIN ADDENTRY(Q, ID[i]);	i番目の要素をエントリに入れる．
Q. LEV:=LEVEL[i] ;	
TESTRELATION:	QがPの兄弟であるか,息子であるか,あるいは直接関係ないかをきめるCASE文．
CASE SIGN(P. LEV−Q. LEV)+2 OF	
BEGIN	
BEGIN IF P. SON≠0 THEN STOP;	息子のとき，Pがすでに息子を持っていれば，これは誤り．さもなければ，父ー息子の関係をそれぞれの欄に記入．
Q. FATHER:=P; P. SON:=Q	
END;	
BEGIN Q. FATHER:=P. FATHER;	兄弟．たがいに同じ父を持つ．
P. BROTHER:=Q	QはPの次の弟．
END;	
BEGIN P:=P. FATHER;	直接関係がない．Pが父をもたない時は誤り．さもなければ，QがPの父と関係があるかどうか調べる．
IF P = 0 THEN ERROR;	
GOTO TESTRELATION;	
END;	
END;	CASE文の結び,
P:=Q;	PにQを与えて，次の構成要素の処理を続ける．
END;	
IF P0. BROTHER ≠ 0 THEN ERROR;	全ての要素の処理終了．第1要素が兄弟を持たないことを確認する必要がある．
END	

図 10.2 構造体のための記号表を作成するプログラム

A_n. …. A_1. A_0 の処理

A_0 という名前のエントリは，複数個存在する可能性がある．問題は，この中から，A_1, A_2, …, A_n によってただ一つに限定されたエントリを見出すことである．A_0 に対するエントリは全て調べなければならない．その一つの検査の第 i 段において，A_{i-1} のエントリから始めて FATHER 欄を上にたどって集団名 A_i を探す，このようにして，最後に A_n が見付かると，検査を開始した A_0 の参照が見付かったことになる．この参照があいまいでないかどうかを確かめるために A_0 に対する全てのエントリに対して検査を行わなければならない．

下にあげたプログラムでは，名前 A_i は，下限0の文字型配列 A に入っており，変数 n で，参照 A_n. …. A_0 の中に含まれる名前の数を示すことにしている．プログラムの出口で，Q には参照される要素の番地が入っている．そのようなデータがなければ0が入っている．

A_n. … A_1. A_0 が参照する要素を探す

BEGIN POINTER P, Q, S;	P は A_0 のエントリへのポインタ，
INTEGER i;	Q には参照される要素の番地が入る，S
P:=“名前 A[0] の最初のエントリの番地”;	は中間結果用，i は 0(A_0) から n(A_n)まで
Q:=0;	計数するため．
WHILE P ≠ 0 DO	A_0 という名前の全てのエントリを一つず
BEGIN S:=P;	つたどる．
FOR i:=1 STEP 1 UNTIL n DO	FATHER 欄のくさりを上にたどって，
BEGIN LOOP: S:=S. FATHER;	A_1, A_2, …, A_n を順に探す．
IF S = 0 THEN GO TO TRYNEWP;	木の根に到達した．
IF S. ID ≠ A[i]	探しているものでなければ，さらに
THEN GO TO LOOP;	FATHER 欄を上にたどる．
END;	
IF Q ≠ 0 THEN ERROR;	エントリが見つかったとき，Q が0でな
Q:=P;	ければ，それは一義的ではない．
TRYNEWP: P:=P. SAME;	名前A_0を持ったつぎのエントリ番地をP
END	に与えて，これを調べる．
END	

このプログラムはある意味で上向きの探索である，すなわち，各 A_0 から始めて，木の根に向って上ってゆく．ここでは，FATHER 欄と SAME 欄だけしか使用されていないので，SON 欄と BROTHER 欄は，もしほかの目的で使用しないのなら省略することもできる（演習問題3参照）．

言語の中には，構造体の名前を，新しいデータの型のように使ってその型を

持った変数を宣言したり，生成したりできるものがある．ALGOL W では，例えば，

```
RECORD COMPLEX(REAL REALPART;REAL IMAGPART);
COMPLEX B;
```

と書いて，2個の実数型の要素を持つ変数Bを作り出す．任意の参照 A_n. $\cdots$. A_1. A_0 (例えば B. REALPART) において，A_n というのは，構造化データの名前であり（あるいは，そのデータへのポインタであり），この構造化データの型は，A_n から得られる．

このような場合に，下向きの探索を行うことも可能である．これは，A_n によって定まる一意的なエントリから始めて，BROTHER 欄と SON 欄を用いて木を A_{n-1}, $\cdots$, A_1, A_0 と下ってゆく．（このプログラムは，読者に任せよう．）

探索すべき木は，一意的であるけれども，下向き探索法が，上向き探索法より速いという保証はない．それは，各水準において，全ての兄弟を調べなければならないからである．上向き探索法では，FATHER 欄だけを調べればよいのである．ところが下向き法では，全ての息子と全ての兄弟を処理しなければならない．

さらに上向き探索法では，各エントリに，一つ余分の欄をもうけることによって一層スピードを上げることができる．この欄は，構造体の名自身を指す．

MOVE CORRESPONDING 命令

COBOL では，

```
MOVE CORRESPONDING A TO L
```

と書いて，A の基本要素で，L に対応する名前を持っているものの全ての値を，L のそれぞれに移すことができる．例えば，A と L とが本節の最初にあげたような構造であるとすると，上の文は，

```
MOVE A. B. C TO L. B. C
MOVE A. B. D TO L. B. D
```

と等価になる．一方

```
MOVE CORRESPONDING A. B TO L. M
```

も

```
MOVE A. B. C TO L. M. C
MOVE A. B. D TO L. M. D
```

と等価になる．PL/I には，同じような“BY NAME”付き代入文がある．この代入文の実行を支配している規則はつぎのとおりである：X と Y を限定された参照として，MOVE CORRESPONDING X TO Y という文は，X1 あるいは Y1（両者でも可）が基本要素として，

$$X1 = X.A_n.\cdots.A_1.A_0$$
$$Y1 = Y.A_n.\cdots.A_1.A_0$$

となるような X1 と Y1 の全ての可能な組合せに対する MOVE X1 TO Y1 という代入文の系列の省略形である．さらに $A_n.\cdots.A_1.A_0$ という要素のくさりは，いつでも，完全でなければならない．すなわち，$i=1, \cdots, n$ に対して，A_i は必ず A_{i-1} の直接の父でなければならない．

MOVE CORRESPONDING 命令を単純な MOVE 命令の系列に変換する一つの方法はつぎのとおり：変換の任意の段階で命令の系列

```
MOVE CORRESPONDING X1 TO Y1
      :                    :
MOVE CORRESPONDING Xn TO Yn
```

を持っているとしよう．最初は $n=1$ である．各 X_i と Y_i はそれぞれ，記号表のそれぞれのエントリへのポインタで表されており，ここで本質的に必要なのは，ポインタの対 (X_i, Y_i) を要素とする配列である．以下の処理は，上の系列の中に，もはや命令がなくなるまで繰返す．系列の中の最初の命令を，一部ずつ処理して，系列から削除してゆく．

1. 上の系列の中の最初の命令中の X1 または Y1 が，基本要素であれば MOVE X1 TO Y1 という文を出力して，ステップ 3. に進む．
2. X1 のすべての息子 A に対して，Y1 が同じ名前の息子をもっていれば，

   ```
   MOVE CORRESPONDING X1. A TO Y1. A
   ```

   という文を上の系列に追加する．
3. X1 と Y1 を含む最初の文を系列から削除する

この変換は理解しやすいけれども，(X_i, Y_i) の対をおさめる表を必要とする．そこで，以下に，XとYに対する構造体の部分木の節点を，上とは別の方法で走査する，別のプログラムを示す．ポインタP(Q)は，X(Y) の部分木を走査するために使用される．このプログラムは，名前の一意性に関する規則は効力がないという仮定に基いている．COBOL には，構造体のいずれの構成要素も，一意的に参照可能でなければならない，という規則がある（演習問題4参照）．この場合，一たん (P, Q) の対を出力してしまうと，Qのいずれの兄弟がPと同じ名前をもっているかを調べる必要がないのである．したがって，プログラムは，より能率よくなるように，修正することができる（演習問題6参照）．

MOVE CORRESPONDING 処理ルーチン

BEGIN POINTER P, Q, P0; P:=P0:="X のエントリの番地"; Q:="Y のエントリの番地";	P(Q) は，部分木 X(Y) を走査する． P0 は，つねに節点 X を指す．
ELEM: IF P. SON = 0 OR Q. SON = 0 THEN BEGIN output(P, Q); GO TO NEXT; END;	もしPかQのどちらかが基本要素であれば，この対を出力し，次の対の処理に進む．
P:=P. SON; Q:=Q. SON	P, Q の息子の走査にうつる．
TESTEQ: IF P. ID = Q. ID THEN GO TO ELEM;	Q の任意の兄弟がPと同じかどうかを調べて，一致すればELEM に戻る．
NEXT: IF Q. BROTHER ≠ 0 THEN BEGIN Q:=Q. BROTHER; GO TO TESTEQ END;	Q の兄弟を調べる．
BROTHERP: IF P. BROTHER ≠ 0 THEN BEGIN P:=P. BROTHER; Q:=Q.FATHER. SON; GO TO TESTEQ END;	Q のどの兄弟もPと一致しないので，Pのつぎの兄弟をQとその兄弟に対して試す．
P:=P. FATHER; Q:=Q. FATHER; IF P ≠ P0 THEN GO TO BROTHERP; END	Pの兄弟が終ったら，PもQも父に戻り，P0 に戻っていたら，終り．

演習問題 10.2

1. LEV 欄を使わないで記号表を組上げるようにプログラムを変更せよ．

ヒント：LEV 欄を使うのは，P と Q の関係を調べるときに限られる．水

準番号が必要なのは，P と P の FATHER 欄から P のつぎにつながる要素に対してだけである．したがって，それを保持するためにスタックを用いるとよい．

2. LEV 欄を使わないで記号表を組上げるようにプログラムを変更せよ．ただし，ここでは，水準番号を置いておくためのスタックは用いないことにする．**ヒント**：水準番号を必要とするのは，P と P の FATHER 欄からくさりにつながれているリスト上の要素だけである．これらの要素はまだ兄弟を持っていないので BROTHER 欄が空いている．この欄を水準番号をしまうために使用しよう．BROTHER 欄は，兄弟が現れる前に正しく戻しておくことに注意せよ．

3. SON 欄と BROTHER 欄を取除いて記号表を作り上げるようにプログラムを変更せよ．1 A　3 B　2 C のような間違いを検査しなければならないことを忘れないようにせよ．これは，2 C を処理するときに，A がすでに息子を持っているかどうか調べればよい．水準番号自身を調べる方法もある．

4. COBOL では，構造体の全ての要素が一意的に参照可能でなければならない．例えば

　1 X　2 Y　3 Z　2 Z

という構造体では，X の要素 Z は，X. Z が二義的になるので参照不可能である．記号表を作るプログラムを変更してこのような誤りを検出できるようにせよ．

5. 参照 $A_n. \cdots. A_1. A_0$ の A_n がつねに一意的で既知のエントリの名前であると仮定すると，下向き探索法を用いて一意的に参照される要素を見出すプログラムを作れ．

6. 問題4に述べた COBOL の規則を仮定する．MOVE CORRESPONDING 命令を変換するプログラムを変更して，同じ名前を持つ可能性のある2個の要素 P と Q だけが検査されるようにせよ．

第11章

原始プログラムの内部形式

記憶容量が問題になったり，原始プログラム言語が複雑であったり，コンパイラ作成計画の目的が複雑であったりすれば，コンパイラは，まず，原始プログラムを，内部形式に変換する．この形式は，原始プログラムに比べて機械的に取扱いやすくなっている．多くの内部形式では，演算子は，本来，実行される順に現れるようになっている．このようになっていると以後の解析とコード生成に大変役立つ．実際，これらの内部形式は，インタプリタの場合にも使うことができよう．すなわちプログラムが初めから内部形式で書かれていればそれをそのまま実行してしまうようなプログラムを作ることも可能である．

この章では，よく使われる内部形式のいくつかを紹介する．**図11.1** に示す短かい ALGOL プログラムを，それぞれの例の説明に使うことにする．もちろん，原始言語やコンパイラの目的に適った特別の表現法があることは承知しておかねばならない．例えば，FORTRAN では，DIMENSION 文の情報は全て記号表にのせてしまうので目的コードは生成されない．したがってこれに対する内部形式を作る必要はない．また図 11.1 の宣言 `INTEGER K` も内部形式となって現れることはない．これも目的コードは生成されない．

初めの内部形式がどの程度詳細になっていなければならないかも定めなければならない．初めの内部形式にはデータの型変換を行う操作を含んでいなければならないのか．それともこれはもっと後回しにしてよいのか．このほか，for ループは，等価的に代入文と go to 文と条件文の組合せで実現できるが，より上級の形式のままに置いておき，後でそのような形式に変換することもできる．一般に初めの形式は，より簡便でより小さく，一方，より詳細になった翻訳形式は，より最適化しやすくなっており，一般にいろいろな処理を施すのが容易になっている．

```
BEGIN INTEGER K;
      ARRAY A[1:I-J];
      K:=0;
L:    IF I>J
      THEN K:= K+A[I-J]*6
      ELSE BEGIN I:=I+1; I:=I+1; GO TO L END
END
```

図11.1　ALGOL プログラムの例

まず第1節は演算子と被演算子の議論から始めよう．これらをどのように表すことができるか．第2節では，最初の内部形式として，ポーランド記法について述べる．つぎに 4-組（3節），3-組と木と間接 3-組（4節），そしてプログラムのグラフ表現（5節）を述べる．

これらの形式を，全く別々に使っているようなコンパイラはほとんどないことを知ってほしい．これらのいくつかを混成して使っているのが普通であり，それは，コンパイラ設計者の気まぐれや，先入観に依ることが多い．

11.1　演算子と被演算子

全ての内部形式には，共通に存在するものが二つある．すなわち，演算子と被演算子である．これらの演算子と被演算子をどう結合するかというところがそれぞれの内部形式で違っている．ここの議論では，＋，－，/，*，BR (branch，分岐）などの演算子を用いる．もちろん計算機の内部では，これらは，整数で表している．われわれは，＋を 4，－を 5，/ を6などとしておこう．

演算子と被演算子が任意の順序で現れるならば，両者を区別する必要があるかもしれない．この場合，被演算子は，記憶場所を2語とり，最初の語には，演算子を示す整数とは異なった整数が入り，あとの語には，被演算子自身が入るようにする（これらの2語は，もちろん，1記憶単位につめ合されることもある）．ここで扱う被演算子の型は，変数や手続きなどの単純な名前や，定数や，コンパイラが生成した中間結果用の変数，それに添字付変数である．もし全ての名前や定数が，同じ表の中におさめられているのならば，添字付変数を除く，全ての被演算子は，それが記述されている記号表のエントリへのポインタで表すことができる．もし，被演算子の記述が，いくつかの表にわたってい

るのならば，被演算子の第1語に，別の型の被演算子であることを示す整数を置く必要がある（11.2 節参照）.

被演算子欄には，特別な処置なしに，間接番地を指定することもできる．すなわち，参照されている値は実際に必要とされている値の番地であることを被演算子欄に示すことができる.

これは次のように，添字付変数を記述するのに役立つ．添字付変数 A[*i*, *j*, …, *k*] の VARPART (8.4.5 参照) の計算を一連の演算で記述する．つぎは，添字用の特別な演算 SUBS である．この演算の被演算子は，配列名，VARPART および中間結果用番地（T とする．演算の結果，配列要素の番地をこれにしまう）である．A[*i*, *j*, …, *k*] は T を通じて参照する.

添字付変数を取扱う第二の方法は，配列名と VARPART の値の両方を1個の被演算子の中に書くものである．これを図 11.2 に示す．この被演算子はほかのものより大きな場所が必要であるが，この内部表現は，良い目的コードを生成しようというときになると，ずっと効率のよいものを作るのに役立つ．これは後の章で明らかにしよう.

以後添字付変数 A[*i*, *j*, …, *k*] が被演算子のときは A[T] という記号で表すことにする．ここで A は配列名，T は VARPART の値が入っている変数である．もちろん，これ以外の方法も可能で，それは読者がコンパイラを設

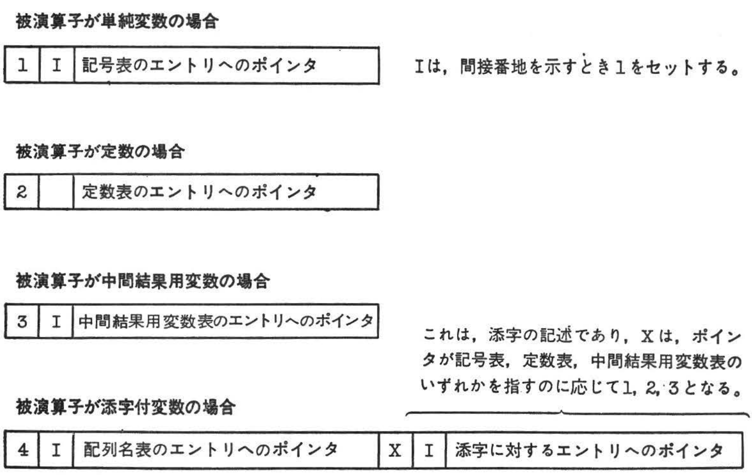


図 11.2 被演算子の内部形式の例

計するときに自分で発見するだろう．

11.2 ポーランド記法

算 術 式

ポーランド記法は，算術式や論理式の表現法の一つであり，これによれば，その中に現れる演算子の評価順序を，単純明確に示すことができる．その上，かっこを使う必要がない．この記法は，ポーランドの論理学者 J. Lukasiewicz が最初に使用したものであり，演算子が被演算子（複数）の直後にくる．したがって，**接尾記法** (suffix notation) とか**後置記法** (postfix notation) などと呼ばれることもある．例えば A∗B は，AB∗；A∗B＋C は AB∗C＋；A∗(B＋C/D) は ABCD/＋∗；そして A∗B＋C∗D は AB∗CD∗＋ と書くことができる．

第 12 章で構文認識アルゴリズムと意味付けルーチンを使って，挿入（インフィクス）記法による算術式をポーランド記法に機械的に変換する方法を説明しよう．ここでは，ポーランド記法に関する以下の規則が，手で変換を行う際に役立つだろう．

1. 挿入（インフィクス）記法でもポーランド記法でも，名前が現れる順序は同じである．
2. ポーランド記法では，演算子は現れる順（左から右）に評価される．
3. 演算子はそれに対応する被演算子（複数）の直後に現れる．したがって，つぎのような構文規則を書くことができる．

〈被演算子〉 ::= 〈名前〉|〈被演算子〉〈被演算子〉〈演算子〉
〈演算子〉　::= ＋|−|/|∗|…

負記号を示す − や，そのほかの同じような単項演算子の取扱いには，2 通りの方法がありうる．一つは，2 項演算子のように書く方法である．すなわち −B は 0−B と書く．もう一方は，− のかわりに新しい記号，例えば @ を導入し，〈被演算子〉 ::= 〈被演算子〉 @ という構文規則を付加えるものである．すると A＋(−B＋C∗D) は AB@CD∗＋＋ と書くことができよう．

われわれは，後置（ポストフィクス）記法と同様に，演算子が被演算子の前にくる前置（プレフィクス）記

法というものも規定することができる．したがって，われわれには3種の記法，前置(プレフィクス)，挿入(インフィクス)（普通に使う記法で演算子が被演算子の間にくる），および後置(ポストフィクス)記法があることになる．普通は，挿入記法を使っているが，算術式を機械的に評価するときには，後置記法が最も扱い易い．以下で評価の方法を調べよう．

算術式の評価

ポーランド記法で表した算術式の評価は，スタックの助けを借りれば，左から右への1回の走査で行うことができる．このスタックには，走査の中で出会った全ての被演算子と，何らかの演算によって生成された全ての結果が保持される．まず最も左側の記号を取り出して処理し，つぎにその右に進んでそれを処理し・・・というように進行する．処理の方法はつぎのようになる．

1. 走査された記号が名前か定数であれば，その値をスタックにのせつぎの記号を走査する．

   これは，〈被演算子〉::=〈名前〉という規則を使うことに対応する．
2. 走査された記号が，2項演算子であれば，それをスタックの上から2個の被演算子に適用し，結果をスタックに置換える．これは，〈被演算子〉::=〈被演算子〉〈被演算子〉〈演算子〉という規則を使用することと意味的に等価である．
3. 走査された記号が単項演算子であれば，それをスタックの一番上の被演算子に適用し，結果をスタックに置換える．これは，〈被演算子〉::=〈被演算子〉〈演算子〉という規則を使うことと意味的に等価である．

このアルゴリズムを AB@CD∗＋＋(普通の表記法で A＋(−B＋C∗D)に適用したものを **図11.3** に示した．スタック上の値は | で分離している．

ポーランド記法における演算子の拡張

ポーランド記法の拡張は，非常に容易である．演算子は，それに対応する被演算子（複数）の直後に置くという規則だけに従えばよい．例えば，代入文の普通の規則〈変数〉:＝〈式〉は，〈変数〉〈式〉:＝ という規則にすればよい．完全な代入文の例として A:＝B∗C＋D は ABC∗D＋:＝ となる．しかしながら，:＝ という演算子の評価は慎重に行わなければならない．:＝ が評価されると，もはや，結果の値というものがないのだから，〈変数〉も〈式〉

ステップ	文字列と走査中の記号	スタックの内容（処理前）	使用した規則	スタックの内容（処理後）
1	AB@CD*++		1	A
2	AB@CD*++	A	1	A|B
3	AB@CD*++	A|B	3	A|−B
4	AB@CD*++	A|−B	1	A|−B|C
5	AB@CD*++	A|−B|C	1	A|−B|C|D
6	AB@CD*++	A|−B|C|D	2	A|−B|C*D
7	AB@CD*++	A|−B|C*D	2	A|−B+C*D
8	AB@CD*++	A|−B+C*D	2	A+(−B+C*D)

図11.3　AB@DC*++ の評価

もスタックから取除かなければならない．この点で，:= は普通の算術用2項演算子とは異なっている．第二に，〈式〉の値を〈変数〉に代入するのであるから，スタックの上で必要なのは，〈変数〉の値ではなくて，番地である．後にも同じような問題をもった別の被演算子に出会うことになろう．

ここで，分岐（GOTO 文）と条件文，配列の宣言と添字をポーランド記法で表記する方法を説明しよう．新しい演算子をポーランド記法に埋込む方法の説明としてこれで十分なはずである．

GO TO A という分岐は "A BRL" と書こう．ここで名札 A は，記号表の中の A のエントリへのポインタで表し，演算子 BRL は，名札への分岐を意味している．

条件分岐は，構文規則

〈被演算子^1〉〈被演算子^2〉BP

を持っており，第1被演算子は，算術式の値であり，第2被演算子は，ポーランド記法による記号列の記号の番号あるいは場所を示している．もし，第1被演算子の値が正であれば，次に走査すべき記号は，第2被演算子が示している記号になる．0 または負であれば，走査はそのまま先へ進む．このほか，BM：負のとき分岐，BZ：0のとき分岐，BMZ：負または0のとき分岐，BPZ：正または0のとき分岐，などの演算を導入できる．

条件文

IF〈式〉THEN〈文^1〉ELSE〈文^2〉

は，つぎのような形式になるだろう．

〈式〉〈op^1〉 BZ 〈$文^1$〉〈op^2〉 BR 〈$文^2$〉

ここで式の値が 0 を偽，≠0 を真ということにしておこう．このほかの要素はつぎのような意味を持っている．

1. 〈op^1〉は〈$文^2$〉の始まる記号の番号または場所を示す定数．
2. BZ は，2個の被演算子——〈式〉〈op^1〉——を持つ演算子で，その意味はつぎのとおり：もし〈式〉がゼロならば（そしてそのときに限り）評価の順序を〈op^1〉で示す番号から始まる記号に変更する．
3. 〈op^2〉は〈$文^2$〉に続く記号の番号である．
4. BR (BRanch) は，被演算子を1個——〈op^2〉——持つ演算子で，その意味は，評価の順序を〈op^2〉から始まる記号からに変更するものである．

BZ は，2項演算子で演算結果を作り出さないことに注意．BZ を評価するとき，スタックの上から2個の要素には，〈式〉と〈op^1〉がのっている．したがって BZ の評価の過程の一部には，スタック上のこれら2個の要素を取除くことがある．同様なことは，無条件分岐 BR についてもいえる．これまで，無条件分岐の演算子が2個あったことも注意しよう．〈名札〉BRL は，原始プログラム中の GO TO 文のために使用されるもので，名札のところには，記号表のエントリの番地が必要である．一方 BR は，コンパイラが内部で作り出した分岐である．

条件文の被演算子は，〈式〉，〈$文^1$〉，〈$文^2$〉であり，ポーランド記法では，

〈式〉〈$文^1$〉〈$文^2$〉 IF

と書けよう．ところが，このような文を含んだポーランド記法の記号列を評価しようとするとどんなことが起るか考えてみよう．演算子 IF を評価する時点では，すでに3個の被演算子が全て評価，あるいは実行されている．われわれの評価は，分岐命令がなければ，左から右の順で行っていることを思い出そう．われわれの使命は，左から右に評価したときに，条件文が望みどおりに実行されるような構造を作ることだったのである．

ALGOL の配列宣言 ARRAY A[L1 : U1, ⋯, L_n : U_n] は L1 U1⋯L_n U_n A ADEC のように表現することができよう．ここで ADEC だけが演算子である．この演算子は，添字の数に応じて可変個の被演算子を持っている．被演算子 A は，もちろん，記号表の A のエントリへのポインタである．ADEC

```
(1)  BLOCK 1 I J - A ADEC K 0 :=
(11) I J - 29 BMZ
(16) K K I J - A SUBS 6 * + := 41 BR
(29) I I 1 + := I I 1 + := L BRL
(41) BLCKEND
```

図 11.4 図 11.1 のポーランド記法による表現

を評価するときは，この A のエントリを参照し，A の次元数を知って初めて，ADEC の被演算子の数を知るのである.

ここで，名前の順序が変っていたことに注意. ADEC の引数の数は可変なので，最後の引数が A であると記号表の A のエントリから，ADEC の引数の数を決定できるからである. もし，A L1 U1…L_n U_n ADEC という順になっていると，引数の個数を知る方法がなくなるので，配列の次元数などを，さらに余分の引数として ADEC の直前に挿入しなければならなくなる.

添字付変数 A[〈式〉, …, 〈式〉] は，同じように 〈式〉…〈式〉 A SUBS と表す. SUBS を評価すると，配列名の表と添字式を使って，配列要素の番地が計算される. つぎに，これらの被演算子を全てスタックから取除き，新たに，この配列要素の型と番地を示す値がスタックにのせられる.

図 11.1 のプログラムを考えてみよう. このプログラムをポーランド記法で表すと**図 11.4** のように書けるだろう. この図では，被演算子を持たない二つの演算子 BLOCK (ブロックを開設する演算子) と BLCKEND (ブロックを閉じる演算子がさらに加わっている.

SUBS 演算子は配列の添字計算を実現するために使われたことに注意. これはポーランド記法で用いる方が容易である. 次節では，番地計算の別の方法の例をみることにしよう. 上の図 11.4 が，計算機の中で，あるいは適当な高水準言語で，実際にどんな具合になっているかを調べてみることは役に立つかもしれない. **図 11.5** にこれを示す. 図 11.4 の各記号が各行に 1 個ずつ置かれている. 演算子は，記憶場所を 1 個使いつぎのように表される： 6 = SUBS, 7 = :=, 8 = BMZ, 9 = BR, 10 = BRL, 11 = BLOCK, 12 = BLCKEND, 13 = ADEC, 14 = +, 15 = *, 16 = − である. もし記号が，定数であれば，2 記憶単位が使用される. 第1語には "定数" を表すために "1", 第2語には定数自身が入る. 同様にもし記号が，名前であれば，第 1 語には "名前" を示す "2" が第 2 語には，記号表のこの名前のエントリへのポインタが入っている. 図 11.4 と 図 11.5 で番号が変っていることに注意. 前者では記号に

語番号	語の内容		記　号	語番号	語の内容		記　号	記号表のエントリ	
1	11		BLOCK	36	6		SUBS		
2	1	1	1	37	1	6	6	1	I
4	2	1	I	39	15		*		
6	2	2	J	40	14		+	2	J
8	16		−	41	7		:=		
9	2	3	A	42	1	64	64	3	A
11	13		ADEC	44	9		BR		
12	2	4	K	45	2	1	I	4	K
14	1	0	0	47	2	1	I		
16	7		:=	49	1	1	1	5	L 17
17	2	1	I	51	14		+		
19	2	2	J	52	7		:=		
21	16		−	53	2	1	I		
22	1	45	45	55	2	1	I		
24	8		BMZ	57	1	1	1		
25	2	4	K	59	14		+		
27	2	4	K	60	7		:=		
29	2	1	I	61	2	5	L		
31	2	2	J	63	10		BRL		
33	16		−	64	12		BLCKEND		
34	2	3	A						

図11.5　図11.4の内部表現

番号をつけたが，後者では，語（記憶単位）に番号を付けた†.

演習問題 11.2

1. つぎの算術式をポーランド記号列に変換せよ：

   1 + 5 * (6 + 8/4)，− 3 + 6 * (− 5 + 6)，25/(5/1 + 6 − 5).

2. 本節に述べた方法を用いて，問題1で得られたポーランド記号列を評価せよ.

3. つぎの文をポーランド記法で表せ.

   ⋯ FOR I := 1 STEP 1 UNTIL 10 DO A[I] := 1;

   (2) CASE 〈式〉 OF BEGIN S1; S2; ⋯; SN END;

   (3) IF 〈式〉 THEN S1;

   (4) DO 〈文の番号〉 〈変数¹〉 = 〈変数²〉, 〈変数³〉 (FORTRAN の DO ル

† **訳注　図11.4** の (11)，(29)，(41) が **図11.5** でそれぞれ (17)，(45)，(64) に対応している. また 11.1 節 (**図11.2**) では，定数を 2，単純変数を 1 としていたので要注意.

ープ).

11.3 4-組

算術式用の 4-組

単一の2項演算に対する便利な形式は 4-組である.

(〈演算子〉, 〈被演算子¹〉, 〈被演算子²〉, 〈結果〉)

ここで, 〈被演算子¹〉 と 〈被演算子²〉 は 2 個の被演算子を示し, 〈結果〉 は結果を示す. したがって A*B は

*, A, B, T

と表すことができよう. ここで T は, A*B の結果を代入する適当な変数である. このようにして, A*B+C*D はつぎのような系列になる.

*, A, B, T_1
*, C, D, T_2
+, T_1, T_2, T_3

ここで注目すべき重要な点は, 普通の形式 A*B+C*D に対比して 4-組は, 評価される順に, 並んでいるということである. 単項演算子をこの形式で書くと, 〈被演算子²〉 が空になる. したがって −A は, "−, A, , T" となり, これは "−A の値を T に代入せよ" という意味になる. ポーランド記法でしたように, 単項演算子 "−" と 2 項演算子 "−" を区別するために, 別の演算子を用いたくなるような場合もいくつかある.

4-組の拡張

4-組をほかの演算子にまで拡張することはかなり容易である. ここでは, 以後に必要となる演算子の一覧表をあげることにしよう. この表で P, P1, P2, P3, P4 というのは, 図 11.2 にあげた形式のいずれかをとる被演算子である. 図 11.6 に, 図 11.1 のプログラムを 4-組で記述したものを示す.

	演算子	被演算子	4-組の意味
1.	BR	*i*	4-組 *i* への分岐
2.	BZ[BP, BM]	*i*, P	P の値がゼロ [正, 負] に応じて, 4-組 *i* に分岐.

3.	BG[BL, BE]	*i*, P1, P2	P1 の値が P2 の値より大きい［小さい，等しい］ならば，4-組 *i* に分岐.
4.	BRL	P	記号表の P のエントリで記述されている 4-組に分岐.
5.	+[*, /, −]	P1, P2, P3	P1 の値と P2 の値を加えて［乗じて，除して，減じて］結果を P3 にしまう.
6.	:=	P1, , P3	P1 の値を P3 にしまう.
7.	CVRI	P1, , P3	P1 の値を実数から整数に変換して P3 にしまう.
8.	CVIR	P1, , P3	整数から実数に変換.
9.	BLOCK		ALGOL のブロックを開設する.
10.	BLCKEND		ALGOL のブロックを閉じる.
11.	BOUNDS	P1, P2	P1 と P2 は配列の上限と下限を示す.
12.	ADEC	P1	P1 という ALGOL 配列の宣言，もしこの配列の次元が n であれば，この前に，各次元の上限と下限を示す n 個の BOUNDS 演算子が現れている.

2次元の配列の要素の番地計算法を示すために，C := A[i, B[j]] という代入文を考えよう．もし d_1 が A の第2添字の幅を示すとすれば，

```
*, i, d1, T1
+, T1, B[j], T2
:=, A[T2], , C
```

```
(1)  BLOCK                (10) +  K, T4, T5
(2)  −  I, J, T1          (11) :=  T5, , K
(3)  BOUNDS 1, T1         (12) BR 18
(4)  ADEC A               (13) +  I, 1, T6
(5)  :=  0, , K           (14) :=  T6, , I
(6)  −  I, J, T2          (15) +  I, 1, T7
(7)  BMZ  13, T2          (16) :=  T7, , I
(8)  −  I, J, T3          (17) BRL L
(9)  *  A[T3], 6, T4      (18) BLCKEND
```

図11・6 図11.1 のプログラムの 4-組による表現

と書けるだろう．

演習問題　11.3

1. 問題 11.2.3 の文を 4-組表現で記述せよ．

2. 手続き呼出し A(X, Y, ···, Z) に対する 4-組表現を設計せよ．引数には，配列名，定数，変数，式などがありうる．引数の中に手続き（関数）呼出しを含んでもよいことに注意せよ．

11.4　3-組，木，間接 3-組

3-組

4-組の持っている欠点の一つは，記述しなければならない中間結果用変数の個数にある．3-組を使うとこの問題が除かれる．3-組はつぎの形をしている．

〈演算子〉〈被演算子¹〉〈被演算子²〉

ここには，結果をしまう欄はない．その代り，後でこの演算の結果を被演算子とするようなときは，この演算の番号を直接参照する．すなわち，

```
A + B * C は
    (1)  *  B, C
    (2)  +  A, (1)
```

と表す．この表記法では，(1) は 3-組の結果を参照するもので，定数 1 ではない．1 + B * C は

```
    (1)  *  B, C
    (2)  +  1, (1)
```

と表すことになる．もちろん，機械の内部表現としては，この新しい被演算子の型——ほかの 3-組 の参照——と別の型とを区別するのに，被演算子の第 1 語に新しい整数を持ってこなければならない．

前節と同じような演算子の表記法を用いて，図 11.1 のプログラムをこの表記法で表したものを **図 11.7** に示す．このプログラムの表現では，4-組の場

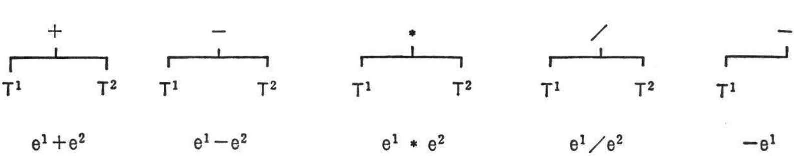
```
(1) BLOCK                 (10) +  K, (9)
(2) -  I, J               (11) :=  (10), K
(3) BOUNDS 1, (2)         (12) BR (18)
(4) ADEC A                (13) +  I, 1
(5) :=  0, K              (14) :=  (13), I
(6) -  I, J               (15) +  I, 1
(7) BMZ (13), (6)         (16) :=  (15), I
(8) -  I, J               (17) BRL L
(9) *  A[(8)], 6          (18) BLCKEND
```

図11.7　図 11.1 のプログラムの 3-組による表現

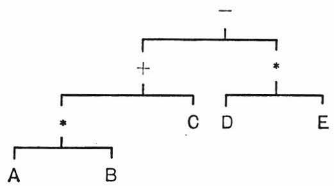
合と同数の 3-組が必要となっていることに注意．しかしながら，ここでは，中間結果用変数はもはや全く不要で，第二に 3-組の方が 4-組より場所をとらない．これらの 3-組が処理されるときに，他の演算の結果である被演算子の記述をたどらねばならぬのはもちろんであるが，それは今なお値が処理中のものだけにかぎられている．例えば，図 11.7 の 3-組 (2) を処理するときに，その結果の記述を作り出すが，3-組 (4) を処理した後には，この記述はもはや必要なく，捨ててしまうことができる．

木

普通の算術式に対する**木** (tree) をつぎのように定義しよう．単純変数や定数用の木はそのまま単純変数や定数である．もし式 e^1 と e^2 が木 T^1, T^2 に対応すれば，式 e^1+e^2, e^1-e^2, e^1*e^2, e^1/e^2, $-e^1$ に対する木は，それぞれつぎのようになる：



例えば，A＊B＋C－D＊E の木は，つぎの左側のようになる：



(1)　(* A, B)
(2)　(+ (1), C)
(3)　(* D, E)
(4)　(− (2), (3))

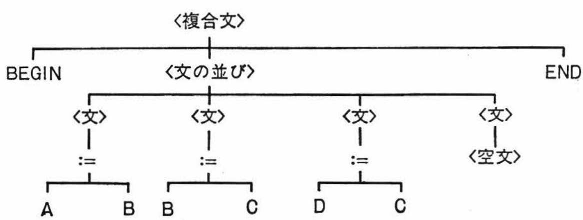
さて，任意の算術式に対する 3-組は，上図の木の右側に書いたように，木を直接表現しているものとみなすこともできる．最後の 3-組 (4) が木の根を表している．それぞれの 3-組が部分木を表す．3-組の中で演算子がそれぞれの部分木の根となっている．一方それぞれの被演算子は，端点を表す変数名か，別の部分木を示す番号である．

3-組から目的コードを生成する方法は，これらの 3-組の系列を，その順に実行される演算の系列とみなすか，あるいは一つの木とみなすかにかかっている．これは第 17 章で調べよう．

文や，ブロックや宣言などということになると，3-組は，これまでと同じような意味で完全な木の形にならない．なんとなれば，別々の文や宣言の間の結合が陽に与えられていないからである．例えば，複合文

```
BEGIN A := C; B := C; D := C; END
```

に対する木と 3 -組はつぎのようにしてよかろう：



(1)　(:= B, A)
(2)　(:= C, B)
(3)　(:= C, D)

木では，各文と直接関係（ポインタ）をもっているのに，3-組表現では，これらの関係は陽に現れない．

間接 3-組

目的コードの最適化（第 18 章）を行うには，普通，原始プログラムから，適当な演算を取除いたり，別の適当な演算をそのプログラム内の，他の場所に

移すことなどが必要である．4-組でこれを行うのはたやすいが，3-組では，相互に参照し合っているので，4-組よりむずかしくなる．この問題を解決するために，表を二つ用意する．一つは，TRIPLE で 3-組自身をしまうもの，もう一つは，OPER で，これには，実行される順序に，3-組の参照の系列がのせてある．例えば，A := B*C; B := B*C は，つぎのようになる．

OPER 表		TRIPLE 表
1.	(1)	(1) * B, C
2.	(2)	(2) := (1), A
3.	(1)	(3) := (1), B
4.	(3)	

これによって，3-組とその相互の参照関係をこわさずに，OPER 表の上の演算を並べ直すことができる．

全く同じ 3-組は，TRIPLE 表にはただ 1 度だけ現れればよいことに注意．要素が全て等しい二つの 3-組は，同じものとみなされる．異なるブロックで宣言された，別の変数 B に対する被演算子はもちろん別物である．

一般に，プログラムの中には，同等な 3-組がたくさんある．とくに添字付変数がある場合にはそのようになる．そこで 3-組の数が OPER 表の演算の数より少なくなる．このことは，3-組を一つ表にのせるとき，それがすでに表中にあるかどうか表全体を探索しなければならず，これには時間がかかる．ハッシュ技法はここでも非常に有効であろう．

図 11.8 は，図 11.1 のプログラムを間接 3-組で表したものである．

被演算子が 3-組を参照しているときは，要注意である．3-組は，あちこち

OPER

1.	(1)	10.	(8)
2.	(2)	11.	(9)
3.	(3)	12.	(10)
4.	(4)	13.	(11)
5.	(5)	14.	(12)
6.	(2)	15.	(11)
7.	(6)	16.	(12)
8.	(2)	17.	(13)
9.	(7)	18.	(14)

TRIPLE

(1)	BLOCK		(8)	+	K, (7)
(2)	−	I, J	(9)	:=	(8), K
(3)	BOUNDS	1, (2)	(10)	BR	18
(4)	ADEC	A	(11)	+	I, 1
(5)	:=	0, K	(12)	:=	(11), I
(6)	BMZ	13, (2)	(13)	BRL	L
(7)	*	A[(2)], 6	(14)	BLCKEND	

図 11.8 図 11.1 のプログラムの間接 3-組による表現

別々の場所で実行されるかもしれないし，その結果を参照することと，その3-組を正しく実行することとが結び付けられていなければならない．例えば，図11.8で，3-組（+，I，1）は，OPER表に従えば演算番号13と15で2度実行される．演算番号16において，3-組12を評価する際に3-組11の2度目の参照が生ずる．これは，演算番号14で3-組12を評価した結果と関連づけられねばならない．これは目的コードの最適化のために，演算が，あちこち動かされると，もっと問題になってくる．

11.5 ブロック

目的コードの最適化を行うときには，プログラムを"基本ブロック"（これは，ALGOLのブロックとは違う）に分割し，これらのブロックでどのような関係にあるかを示す"プログラム・グラフ"を作ると便利なことがよくある．基本ブロックとは，入口と出口を1個づつ持った，演算の系列であって，入口と出口はそれぞれ最初と最後の演算に対応する．さらに，これらの演算は，その順に実行され，ブロック中には分岐を含まない．例えば，**図11.9a**には図11.1のプログラムの基本ブロックを4-組を用いて示した（間接3-組を用いてもよい）．ここで，分岐の被演算子は4-組の番号ではなく，ブロックの番号である．ブロック2の4-組BMZは，つぎのような意味である．もしT2が負またはゼロであれば，ブロック4へ，さもなければブロック3へ分岐せよ．一般に，ブロックの最後の演算は，任意個のブロックへの分岐である．もし後続のブロックがただ一つしかなければ，そこへの分岐をそのブロックの最後に陽に書く必要はないことにも注意．このような分岐は，プログラム・グラフ上に，

```
1. BLOCK                 3. -    I, J, T3
   -    I, J, T1            *    A[T3], 6, T4
   BOUNDS 1, T1             +    K, T4, T5
   ADEC A                   :=   T5,, K
   :=   0,, K
                         4. +    I, 1, T6
                            :=   T6,, I
                            +    I, 1, T7
2. -    I, J, T2            :=   T7,, I
   BMZ 4, 3, T2
                         5. BLCKEND
```

(a) 基本ブロック

$$\begin{bmatrix} 0&1&0&0&0\\ 0&0&1&1&0\\ 0&0&0&0&1\\ 0&1&0&0&0\\ 0&0&0&0&0 \end{bmatrix}$$

(b) プログラム・グラフ

図11.9　図11.1のプログラムの基本ブロックとプログラム・グラフ

暗黙のうちに与えられている.

プログラム・グラフは，基本ブロックごとに，そこから直接分岐するブロックのリストと，そこに直接分岐してくるブロックのリストを持っている. 図11.9では，これを行列 M で表した. ここで $M[i, j]=1$ はブロック i からブロック j への分岐を示す. 実際，これを表現するには行列よりリスト構造の方が都合がよく，それはそれぞれのブロックに関してもっと情報を盛込みたいし，最適化処理の過程で，基本ブロックは，作り出されたり，消されたりするからである.

基本ブロックの形式と意味は制限されているのでこのようにプログラムを表現すると，基本ブロック内で，命令の列を最適化することができるようになる. また，プログラム・グラフを解析してループを見付けたり，演算をあるブロックから別のブロックへ移したりすることもできるようになる. これは第18章で論じよう.

いろいろな変形も可能である. 例えば，もとのプログラム中のループだけを最適化すべきものとすると，ループの処理を容易にするためにループの中の演算は，ある種のブロックとして当然一まとめにされよう.

11.6 参考文献概説

ポーランド記法や 4-組や 3-組は，数多くのコンパイラの中で内部表現として用いられてきている. ブロックやプログラム・グラフの使用は，目的コードの最適化を容易にするために工夫された. Allen (69) は，間接 3-組を用いてそのような機構を使った. 一方 IBM 360 の FORTRAN H では，同じような機構を用いているが，4-組で行った (Lowry および Medlock (69) 参照).

リスト構造で表した構文の木は，いくつかのトランスレータ作成システムの中で用いられている. 最もよく知られているのは，Brooker と Morris のコンパイラーコンパイラと Reynolds の COGENT である. ALGOL W コンパイラもこれを使用している (Bauer, Becker および Graham (68) 参照).

第12章

意味付けルーチン入門

本章では文法の個々の規則と意味付けルーチンを関連づける考え方を議論する．個々のルーチンと結び付いている規則が，構文還元操作の中に含まれているときに，このルーチンは意味付け処理を行う．12.1節では，上向き解析法を仮定して，算術式を挿入(インフィクス)記法からポーランド記法に変換する問題と一しょにこの考え方を紹介する．12.2節では，算術式から4-組を作り出す方法を説明する．これには，構文の木に意味情報を付加する必要がある．12.3節では，この意味情報の表現方法とコンパイラの中での意味付けルーチンの様子を議論する．最後の12.4節では，下向き解析法を用いたときの意味付け処理を簡単に見てみよう．

本章の目的は，つぎのような構文で表される算術式を別の形式に翻訳する方法を示すことにある．したがって，ここでは，記号表とか，属性とかなどには関心がない．

(12.0.1) $Z ::= E$
$E ::= T | E + T | E - T | - T$
$T ::= F | T * F | T/F$
$F ::= I | (E)$

12.1 ポーランド記法への翻訳

文形式の把手 x が，非終端記号 U に還元されようとするときはいつでも，$U ::= x$ という規則に対応するルーチンが呼ばれるという仕組で構文認識アルゴリズムが働いていると仮定しよう．このルーチンは x 中の記号を意味的に処理して，x に直接関わるポーランド記法の部分系列を出力する．ここでは，どんな認識方法がとられているかは関心がないことに注意しよう．ここで予期し

ているのは

(12.1.1)　把手は，還元操作のたびに必ず還元される．

ということである．すなわち，最も左側の単純句が還元される．(演算子順位法の場合には，わずかな修正が必要であるが，これはここでは関心がない．) 把手が還元されるからには，つぎのような重要な仮定を置くことができる：

(12.1.2)　把手に任意の非終端記号Vが現れると，Vに還元される部分系列に対応するポーランド記法の部分系列がすでに生成されている．

生成されつつあるポーランド記号列は，1次元配列Pにたくわえられると仮定しよう．初期値1の整数型変数 p で，つぎに入るべき配列要素を示す．各配列要素には1個の記号（名前か演算子）を入れることができる．また，ある意味付けルーチンが呼ばれると，このルーチンは，認識アルゴリズムで用いられる構文用スタック S(1), …, S(i) の中にある把手に含まれる記号（複数）を参照することができるということも仮定しよう．

意味付けルーチン

構文規則 $E^1 ::= E^2 + T$ に対応する意味付けルーチンを考えよう．このルーチンが呼ばれるときには，E^2 と T に対するポーランド記号列がすでに生成されていると仮定することができる．実際，E^2 が T のすぐ左側にあるので配列Pには

…〈E^2 に対するコード〉〈T に対するコード〉

が入っている．Tの右側には，ほかにどんなコードも生成されていないことに注意．原始プログラムのTの右側には終端記号だけが存在する．それは，これが左から右への解析だからである．そこで，このルーチンがなすべきことは，ポーランド記号列に＋記号を加えることしかない．このようにして，挿入(インフイクス)記法 $E^2 + T$ から後置(サフイクス)記法 E^2T+ に翻訳する．したがってこの意味付けルーチンは，まさに $P(p) := {}'' + {}''; p := p + 1$ である．

構文規則 $F ::= I$ (I は任意の名前) に対応する意味付けルーチンを考えよう．第11章で与えた，ポーランド記法に関する規則によって，名前はその演算子より前にくること，さらに，挿入記法の場合と同じ順序で名前が現れることである．この規則で行うことは，名前を出力用配列Pに置くことだけである．したがって，意味付けルーチンは

$$P(p) := S(i);\ p := p+1$$

となる．ここで S(*i*) は，スタックの最上段にある記号である．

F ::= (E) に対する意味付けルーチンは，何もする必要がない．ポーランド記法には，かっこは現れないし，被演算子Eに対するコードはすでに生成されている．

このほかのルーチンも同様な方法で構成できる．これらの一覧表を**図 12.1** に与えよう．同じような意味を持ったいくつかの規則に対して，できれば同じルーチンを用いたい．例えば，規則3，4，7および8に対する意味付けルーチンは，ただ1個のルーチン

$$P(p) := S(i-1);\ p := p+1$$

で置換えることができる．

	規　則	意味付けルーチン
(1)	Z ::= E	なし
(2)	E ::= T	なし
(3)	E ::= E+T	P(p) := "+"; p := p+1
(4)	E ::= E−T	P(p) := "−"; p := p+1
(5)	E ::= −T	P(p) := "@"; p := p+1
(6)	T ::= F	なし
(7)	T ::= T*F	P(p) := "*"; p := p+1
(8)	T ::= T/F	P(p) := "/"; p := p+1
(9)	F ::= I	P(p) := S(i); p := p+1
(10)	F ::= (E)	なし

図 12.1　ポーランド記号列を生成する意味付けルーチン

文の構文解析

図 12.2 は，文 A*(B＋C) の構文解析を示している．われわれは，スタック S を使用する認識アルゴリズムと入力記号 R および文の残り $T_k \cdots$ とを仮定してきた．この図は構文解析の各段——還元の過程か R をスタックに入れ入力記号列からつぎの新しい記号 T_k を取ってくる過程かのいずれか——を示している．還元が行われる直前に，意味付けルーチンが呼ばれる．このルーチンはポーランド記号列の一部を生成することもあるし，しないこともある．この記号列は図の最後の欄に示す．

スタック S	R	T_k ・・・	使用した規則の番号	意味付けルーチン	これまでに生成されたポーランド記号列
#	A	＊(B+C)#			
#A	＊	(B+C)#	9	9	A
#F	＊	(B+C)#	6	6	A
#T	＊	(B+C)#			A
#T＊	(	B+C)#			A
#T＊(	B	+C)#			A
#T＊(B	+	C)#	9	9	AB
#T＊(F	+	C)#	6	6	AB
#T＊(T	+	C)#	2	2	AB
#T＊(E	+	C)#			AB
#T＊(E+	C	)#			AB
#T＊(E+C	)	#	9	9	ABC
#T＊(E+F	)	#	6	6	ABC
#T＊(E+T	)	#	3	3	ABC+
#T＊(E	)	#			ABC+
#T＊(E)	#		10	10	ABC+
#T＊F	#		7	7	ABC+＊
#T	#		2	2	ABC+＊
#E	#		1	1	ABC+＊
#Z	#		STOP		ABC+＊

図 12.2 A＊(B+C) の構文解析

議　　論

A＊(B＋C) から ABC＋＊ を生成するのに，面倒なことがたくさんあったと感じるかもしれない．ここでは10個の意味付けルーチンを呼出さねばならなかったが，結局のところそれらの大部分は実は何もしなかった．これは，きっと算術式の特別な場合に対してもっと短かい方法を発見してプログラムできることを示唆している．しかしながら，ここで見ているのは，どんな文法の文が与えられてもそれを構文解析し，その文から何らかの意味を取出さねばならない一般の場合である．われわれには，これを行うための何か組織的な方法が必要で，そうすれば，新しい言語がやってきたときでも，完全にゼロから出発しなくてすむようになる．

われわれは問題を二つの部分——構文と意味——に分離してきた．これによってつぎのような理由で問題が簡単になった．まず第一に，文法を見ると，ある文がどのように構文解析されるか——どんな順序で還元されてゆくか——がわかる．これは，その都度プログラムしなくてよい．われわれは**自動作成機** (constructor) と認識アルゴリズムを一度プログラムしておいて，それを使えばよい．第二の理由は，各生成規則ごとに，意味付けルーチンを一つ書くこと

ができることである．これによって，あらゆることを一時に考えなくても，小さな一つ一つ独立な部分に分けて別々に仕事をするのに役立つ．

最後に，構文や意味に部分的な変更がある場合にも，それに対する規則や意味付けルーチンをわずかに変更するだけでよい．各部分が分離されているのでこのような変更が容易なのである．

演習問題　12.1

1. つぎの各文を，図12.1の意味付けルーチンと（12.0.1）の文法を用いて，構文解析せよ．

   A, A + B, (A + B), (((A))), ((A + B) * C + B) * C.

2. （12.0.1）の文法に，代入文の規則 S :: = I : = E と，READ 文の規則 S :: = READ I を付加えて，これらの文に対するポーランド記法を決定せよ．そしてこれらをポーランド記法の形式に翻訳する意味付けルーチンを書け．

3. 問題2のルーチンを用いて，A : = A + B および READ C を構文解析せよ．

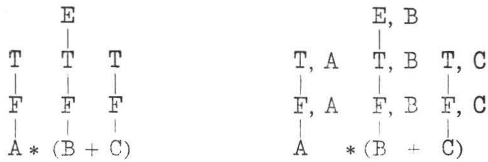
4. （12.0.1）の文法に添字付変数に関する規則（複数）を付加えて，それらに対する意味付けルーチンをそれぞれ書け．

12.2　4-組への翻訳

式 A * (B + C) を構文解析して．

+, B, C, T1
*, A, T1, T2

という4-組を生成する意味付けルーチンを作ることを試みよう．論理的には最初の4-組を生成するのは E :: = E + T という規則に対応する意味付けルーチンである．したがって，上向き構文解析を実行して，この規則に行きつくところまで行ってみよう．解析の各段で生成される文形式を以下のように示す．ここで各段での把手には下線を施してある．図12.3には，ここまでの構文の木が与えてある．



a) 意味付けなしの場合　　　b) 意味付けの場合

図 12.3 部分的な構文の木

$\underline{A} * (B + C)$	$T * (\underline{T} + C)$
$\underline{F} * (B + C)$	$T * (E + \underline{C})$
$T * (\underline{B} + C)$	$T * (E + \underline{F})$
$T * (\underline{F} + C)$	$T * (\underline{E + T})$

つぎのステップは E + T を E に還元することである．同時にその意味付けルーチンが4-組を生成させる．ところが不幸なことに，文形式には，被演算子EとTに対応する具体的な変数名に関する情報が与えられていないので，4-組を生成することができない．この情報はF::＝B と F::＝C という還元を行う際に捨られてしまっていた．

ポーランド記法を生成するときに，何の問題も生じなかった理由は，F::＝B と F::＝C という還元を行うときにBとCの名前を出力として出しておいたからである．4-組を生成するときも，これらの名前は，実際に使用するときまで，どこかにとっておかなければならないし，明らかにこれはF::＝I に対応する意味付けルーチンの中で行わなければならない．

このような情報は，大抵の場合，非終端記号と結び付けておく．したがって，図 12.3 b に示すように，構文解析の進行につれて，構文の木の節点に，これらの意味情報を結び付けておくのがよさそうである．こうすれば，以前より名前をより正確に表すことができることも注意しよう．走査部が名前を認識して二つの引数――記号 I（名前を意味する）と，その名前自身――を返すことを思い出してほしい．I は構文上の記号で，名前自身は，図12.3 b に示すように，I に結び付けられている．

以下で，木の中で非終端記号Uに対応する意味を U. SEM として参照する．つぎに，部分式を区別するために，T1, T2, … という名前を生成する必要があるだろう．このために変数 i を用意し初期値は 0 とする．意味付けルーチン

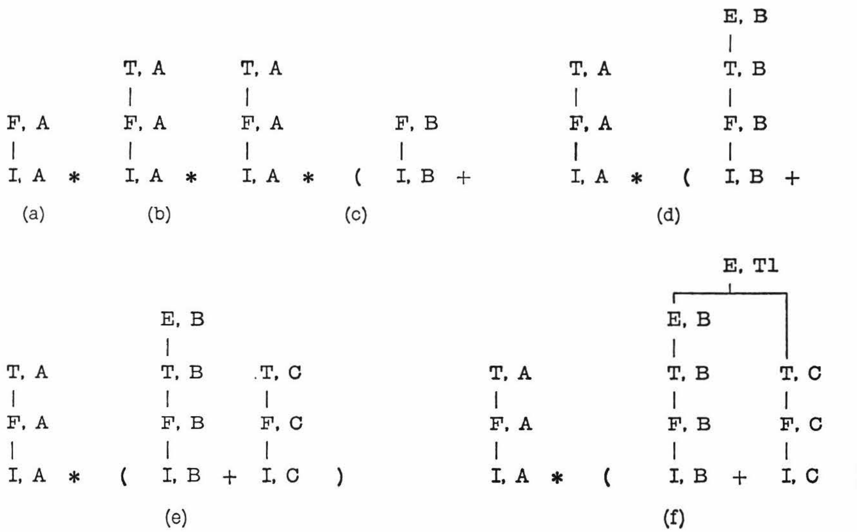


図 12.4　意味を伴った木の部分

のプログラムとしては，"$i := i + 1$; U. SEM : = T$_i$" によって，新しく名前を生成し，それをUに対応づけるのである．

最後に，手続き ENTER (W, X, Y, Z) を呼出して，これら 4 個の引数からなる 4-組を生成して出力とする．

さて，ここで，文字列 A＊(B＋C) を構文解析しながら意味付けルーチンを描き出してみよう．最初の把手はAである（対応する意味はAである）これをFに還元しよう．名前Aは，新しい非終端記号Fに結び付けなければならない．したがって F:: = I に対する意味付けルーチンは

F. SEM : = I. SEM

である．図12.4 **a** にこの手続き実行後の部分的な構文の木を示した．つぎの把手はFである．これはTに還元される．再び，Tは，図12.4 **b** のように今Fに結び付けた名前と結び付けてやらなければならない．これら二つの場合を見ると，これを一般化して，$U :: = V$ という規則に対応する意味付けルーチンは

U. SEM : = V. SEM

規　則		意味付けルーチン
(1)	Z::=E	Z. SEM:=E. SEM.
(2)	E::=T	E. SEM:=T. SEM.
(3)	E^1::=E^2+T	i:=i+1; E^1. SEM:=Ti; ENTER("+", E^2. SEM, T. SEM, E^1. SEM).
(4)	E^1::=E^2−T	i:=i+1; E^1. SEM:=Ti; ENTER("−", E^2. SEM, T. SEM, E^1. SEM).
(5)	E::=−T	i:=i+1; E. SEM:=Ti; ENTER("@", 0, T. SEM, E. SEM).
(6)	T::=F	T. SEM:=F. SEM.
(7)	T^1::=T^2*F	i:=i+1; T^1. SEM:=Ti. ENTER("*", T^2. SEM, F. SEM, T^1. SEM).
(8)	T^1::=T^2/F	i:=i+1; T^1. SEM:=Ti; ENTER("/", T^2. SEM, F. SEM, T^1. SEM).
(9)	F::=I	F. SEM:=I. SEM.
(10)	F::=(E)	F. SEM:=E. SEM.

図 12.5　4-組を生成する意味付けルーチン一覧

であるといってよいようである．これが，実際われわれがこの簡単な文法に対して必要としていたものである．図 **12.4 c** から図 12.4 **e** には，構文解析の進行につれて生ずる二，三の部分木を示している．図 **12.4 e** は把手が E + T の時点における木を描いている．ここで，E^2 + T は規則 E^1 :: = E^2 + T を用いて E^1 に還元してよい．意味付けルーチンは，まず，新しい名前 T_i を作り出し，それを新しい非終端記号 E^1 に結び付けなければならない．つぎに，″ + ″に対する4-組を生成しなければならない．被演算子の名は，E^2. SEM と T. SEM の中にあり，結果を入れる名は E^1. SEM にある．したがって，つぎのルーチンを実行して図 **12.4 f** の構文の木を作り出す．

$i := i + 1$; E^1. SEM : = T_i
ENTER (″ + ″, E^2. SEM, T. SEM, E^1. SEM)

構文解析の残りの部分は読者に任せよう．読者は全部とおしてみて，残された意味付けルーチンを設計してほしい．でき上ったルーチンを図 **12.5** に示したものと比較してみよ．

演習問題　12.2

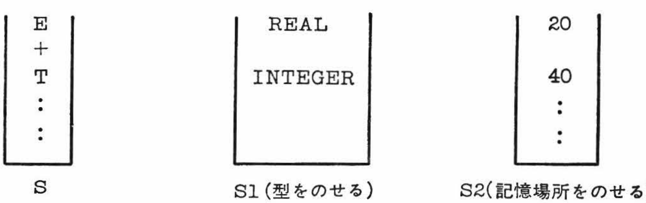
1. つぎの各式を図 12.5 のルーチン群を使って構文解析せよ：A, (A + B), (((A))), ((A + B) * C + B) * C.

2. 二つの規則 S::＝I:＝F と S::＝READ I を文法 (12.0.1) に加えて，これらの新しい文に対する4-組の形式を決定し，それらを生成する意味付けルーチンを書け.

3. A:＝A＋B および READ C を構文解析せよ.

12.3 意味付けルーチンの実現とスタック

一般に，1個の非終端記号に対して複数個の意味情報が存在しうる. しかし，E^1::＝E^2＋T のような規則を適用して，還元を行うときには E^2 や T に関する情報はもはや必要としないことを注意しておこう. 普通の場合現在の文形式の中の非終端記号に関する意味情報だけが必要である. そこで，その情報をたくわえる方法を決定しよう. 現在の文形式は，スタックSに置かれているので，複数個の意味情報用のスタック S1, S2, ･･･ を用意することは容易にできる. これらのスタックは，構文用スタックSと並行して働くものである. 意味付けルーチンはスタック S, S1, S2, ･･･ を全て参照することができる. 例えば，もし S(*i*) に記号Eがおかれていると，S1(*i*), S2(*i*), ･･･ は下図のようにEの型，その実行時の記憶場所，などを置くために使うことができる. 構文の記号と意味属性をのせるために，複数個のスタックを用いるか，複数個の欄を持ったただ1個のスタックを用いるかは，実際にはどうでもよい.



意味との結合

われわれは，個々の意味付けルーチンを，何らかの言語で手続きとしてプログラムすることができる. しかしながら，こうなると構文認識部は，それぞれの手続き名を知っていなければならず，それではあまりにも多くの名前を管理する問題が生じてしまう. そこでそれぞれの規則を正の整数に対応付ける方法をとるとより簡単になる. 例えば，SEMANTICS という名のただ一つの手続

```
PROCEDURE SEMANTICS(r);                    引数は適用される規則の番号
VALUE r; INTEGER r;
CASE r OF                                  実行すべきルーチンの選択
    ;                                      規則1
    ;                                      規則2
    BEGIN P(p):="+"; p:=p+1 END;           規則3
    BEGIN P(p):="−"; p:=p+1 END;           規則4
    BEGIN P(p):="@"; p:=p+1 END;           規則5
    ;                                      規則6
    BEGIN P(p):="*"; p:=p+1 END;           規則7
    BEGIN P(p):="/"; p:=p+1 END;           規則8
    BEGIN P(p):=S1(i); p  p+1 END;         規則9
    ;                                      規則10
ENDCASE
```

図 12.6 文法 (12.0.1) 用の意味付け手続き

きを書いて，その引数の一つは規則の番号とする．還元を行うときには，構文認識部は SEMANTICS を呼び出し，その引数は，還元を行うのに用いる規則の番号とする．図 12.6 は，11.2 節のポーランド記法を生成するために使われるルーチンについてこのことを示している．もしコンパイラが，FORTRAN か ALGOL で書かれていれば，CASE 文を使う代りに，計算型 GO TO 文か，スイッチを使うことができる．構文用スタックと意味用スタックは，FORTRAN では COMMON 領域に，ALGOL では，大域的な変数に置くことができる．一方，引数としてこれらのスタックを，意味付けルーチンに渡すこともできる．

コンパイル時についていえば，最も効率のよい方法は，**自動作成機** (constructor) があって，構文を与えると，コンパイラ作成者が用いている言語で構文認識部を自動的に書き上げてしまうものであろう．作り出されたものには，必要な種々の宣言――いくつかのスタックやそのほかの変数――と，認識部の本体，それに走査部に至るまでの要素が含まれていることになろう．これらの作り出されたものがコンパイラに図 12.7 のようにはめ込まれる．コンパイラとの間にはいくつかの約束事が付随している．すなわち，意味付けルーチンの名札 SEM，意味付けルーチンを終えたときの戻り先の名札，スタックの名前など全て一致していなければならない．意味付けルーチンを実行するには，ほかの手続きは一切呼出さない．全ては，単純な分岐のみで行う．この技法は McKeeman ほか (70) による XPL コンパイラ作成システムで用いられてい

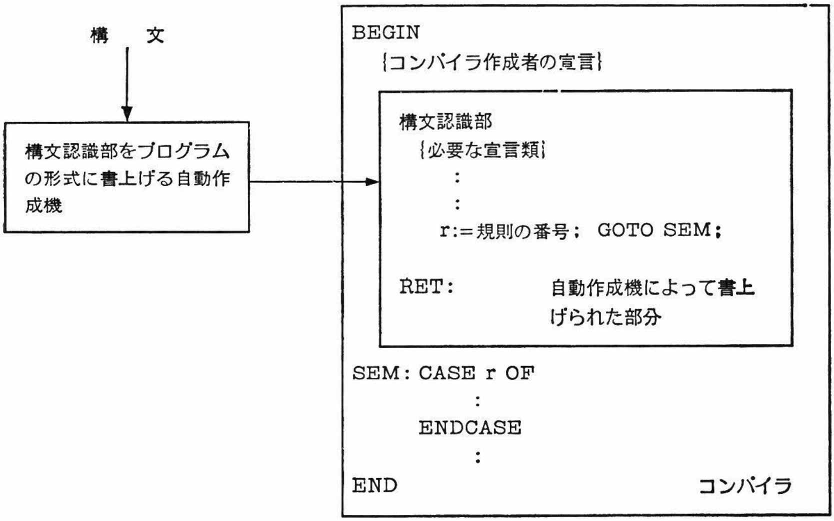


図 12.7　コンパイラの一部を生成する自動作成機

る.

12.4　下向き解析法における意味処理

(12.0.1) の各文を構文解析して翻訳するための，再帰的手続きを作成するために，まず，文法をつぎのように書直す：

Z ::= E
E ::= [−] T {(+ | −) T}
T ::= F{(* | ／) F}
F ::= I | (E)

つぎに，4.3節にならって，つぎの四つの再帰的手続きを書く：

PROCEDURE Z;　　E	規則 Z::=E

PROCEDURE E; BEGIN IF NXTSYMB='−' THEN SCAN; T; WHILE NXTSYMB='+' OR NXTSYMB='−' DO BEGIN SCAN; T END END	規則 E::=[−]T{(+|−)T} 最初が単項演算子−ならば項(term)を取出す. 繰返し +T または −T を探す.

PROCEDURE T; BEGIN F; WHILE NXTSYMB='*' OR NXTSYMB='/' DO BEGIN SCAN; F END END	規則 T::=F{(*|/)F} まず因子(factor)を構文解析する. つぎに，繰返し *F または /F を探す.

PROCEDURE F; BEGIN IF NXTSYMB='I' THEN SCAN ELSE IF NXTSYMB ≠ '(' THEN ERROR ELSE BEGIN SCAN ; E; IF NXTSYMB ≠ ')' THEN ERROR ELSE SCAN END END	規則 F::=I|(E) 名前のチェック. 名前でなければ，(がこなければならない. つづいて E が来る. 最後は閉じかっこでなければならない.

記号を意味的に処理するために，上の手続きのそれに，いくつかの実行文を加えてやる．われわれは4-組を生成することにしよう．ここでは，手続きTとFの処理を説明しよう．残りの二つについては，読者に任せる．

上の手続きでは，構文の木や意味情報用スタックが陽に現れていないことに注意．これらのものを作り上げるためにいくつかの実行文を挿入することもできようが，その必要はない．その代りに，それぞれの手続きでは局所的な変数を用いて，意味情報を受渡すために，仮引数を使用する．

非終端記号Z, E, TあるいはFの意味は，原始プログラム中に現れている名前であるか中間結果用の変数かである．ある非終端記号に対応する句が構文解

析されるときに，この名前がその記号と結び付けられなければならない．そこで，上の手続きの各々に STRING 型の引数 X を設けよう．手続きの実行が終ると，対応する変数の名前を X に返してやる．

以下に改訂した手続き F(X) の例を示す．ここでは，ある名前が走査されると，手続き SCAN が'I'を NXTSYMB に，名前自体を NXTSEM に代入する．この手続きの中で，E(X) を呼び出すと，これは，Eの句に対する変数名を返してくることを注意しておこう．この同じ名前が，Fに対する名前として用いられるのである．

PROCEDURE F(X); STRING X;	F の意味(変数名)を X に返す.
BEGIN	
IF NXTSYMB = 'I' THEN	名前であれば，SCAN によって，その名
BEGIN X:=NXTSEM; SCAN END	前が NXTSEM にすでに置かれている.
ELSE	さもなければ(E)である.
IF NXTSYMB ≠ '(' THEN ERROR	かっこを調べて，E の構文解析をする.
ELSE BEGIN SCAN; E(X);	E に結び付けられた名前はそのままFに
IF NXTSYMB ≠ ')'	結び付けられる！
THEN ERROR ELSE SCAN	
END　　END	

つぎに手続きTを見よう，ここでは，＊と／に対する4-組が生成される．最初に，因子を構文解析し，対応する名前を局所的な変数Yに代入する．つづいて，演算子＊または／と第2被演算子を走査し，これに対する4-組を生成する．4-組を生成するたびに，Y には，4-組の結果を入れる変数名が代入されている．この処理は，＊または／が続くかぎり繰返される．

PROCEDURE T(X); STRING X;	句 T に対する変数名を X に返す.
BEGIN STRING Y, Z, OP;	まず因子を構文解析し，対応する名前を
F(Y); OP:=NXTSYMB;	局所変数 Y に代入する.
WHILE OP = '*' OR OP = '/' DO	演算ごとに，F を呼出してひき続く因子
BEGIN SCAN; F(Z); j:=j+1;	の構文解析をする．中間結果用変数を生
ENTER(OP, Y, Z, Tj);	成して，4-組を生成する.
Y:=Tj; OP:=NXTSYMB;	結果用の名前は Tj である.
END;	結果用の名前 Y を，引数 X にのせて返
X:=Y	す.
END	

以前に示した構文解析アルゴリズムは大抵の場合スタックを用いると云ったが，ここでは，スタックは陽に現れなかった．しかしこれは実行時において手

続き呼出し情報をしまうために用いられるスタックの機構の中で暗黙のうちに使われている（8.8，8.9 節参照）．それぞれの手続きは，種々の値を保持するために局所的な変数を用いている．これらの変数は，その手続きの個々の実行に局所的であることに注意．これがわれわれの望んでいたところである．意味情報は意味情報用のスタックを使って陽に受渡すかわりに，手続きの引数として，受渡される．

13.6節には，コンパイルにおける，下向き解析法の別の使用例が述べてある．

12.5 参考文献概説

普通の（演算子を中に置く）式を別の形式に変換する技法で最も古いものは Rutishauser (52) が与えた．彼の技法は，式全体を繰返し走査し，そのたびに，最も内側の単純な演算を見出して，その内部形式を生成し，それをただ1個の被演算子と置換えるものである．FORTRAN コンパイラの最初のものも，1個の式に対して，何回も走査を行って，内部形式にするものであった (Backus ほか (57) 参照)．Samelson と Bauer (60) が，意味処理を行うルーチンと結び付いた能率のよい（特別な）上向き解析法（推移行列）と呼べるようなものを最初に著した．この論文では，ALGOL 60 の大部分の構成要素の翻訳を議論している．Dijkstra (63) は，式をポーランド記法に翻訳するために（特別な）演算子順位技法を用いることを述べている．これら歴史的に興味深い技法の概説が Shaw (66) にある．

12.1 節と 12.2 節で述べた方法がうまくゆくのは，特定の技法や実現法を用いているからではなくて，これが，まさしく上向き解析アルゴリズムに基いているからにほかならない．

初期のコンパイラでは，構文と意味が，コンパイラごとに，それぞれの方法で混り合っていた．現存のコンパイラでも，なおそういうものが多くある．両者を分離する目的は，構文処理を形式化し，自動化できるようにし，より組織的に取扱うことを可能にすることにある．Irons (61) や Brooker らによる初期のコンパイラ作成システムは，両者を分離した．Brooker らのものは，下向き構文解析の機構を使い，実際に構文の木を組立てているので，意味付けルーチンの中でこの木を取扱うことができる．このルーチンは，FORMAT と名付けられている．意味付けルーチンなる呼び名は Feldman (66) に由来するもので，Wirth と Weber は“解釈規則”と呼んでいる．

第13章

ALGOL型文法要素に対する意味付けルーチン

変数，式，条件文，名札と分岐，それにforループに対する4-組を生成するための意味付けルーチンの説明を行う．これがわかれば，このほかの文法要素に対する意味付けルーチンを作成することに，困難はないはずである．本章では，実行時での変数の記憶場所の割付については考慮しない．これは第14章の話題である．ブロック構造を持たない ALGOL 型言語を仮定し，ここでは，全ての変数は，それが使用される以前に宣言されていなければならない．FORTRANやブロック構造を持った言語に対する注意は適宜行うことにする．

意味付けルーチンは11.3節で述べた記法を用いて4-組を生成する．しかしながら，被演算子は11.2節に述べたようなものではないことに注意してほしい．このような被演算子を作り上げようとすると，あまりに詳細なプログラミングが必要となり，明快な説明を行うという目的からはずれてしまうことになろう．そこで，その代りに，各被演算子は，原始プログラムの全ての変数，一時的変数，定数などの情報を持った，唯一つの記号表のエントリへのポインタとしておこう．さらに上向き解析法を仮定しよう．もっとも本章の最後の節は，論理式に対する下向き解析法を議論している．

本章の目的は，単にコンパイラを書くことではなく，どうやってコンパイラを書くかを説明することにある．読者は，あることがらを何故この方法で行うのかという理由をしっかり理解してほしい．意味に合せて，構文を再編成することに注意してほしいが，新しい構文が現在使用中のある種の認識法に適していることを確かめるという新しく加った問題を議論しようとは思わない．

4-組への変換で述べた方法は，このほかの内部表現への変換や，機械語への変換でも用いられるものが多い．

13.1 意味付けルーチンの用語法

われわれは，プログラム中の全ての名前のための記号表を一つ用い，この表は，下記のような手続き LOOKUP，LOOKUPDEC，INSERT を通して参照される：

1. LOOKUP (NAME, P). 引数 NAME（名前）で，記号表の中に名前の実効エントリを探す．エントリの番地をPに代入する．もしそのようなエントリがなければPには0が入る．

2. LOOKUPDEC (NAME, P). LOOKUP と同様であるが，これは，宣言用である．ブロック構造を持った言語に対しては，このルーチンは，現在のブロックの中の名前だけを見ている．
3. INSERT (NAME, P). 記号表に新しいエントリを設ける．引数は NAME であり，P には，そのエントリの番地が代入されて戻る．

これらの手続き呼出しは，FORTRAN やそのほか大抵の言語をコンパイルするのに用いることができよう．特定の手続きの操作は，記号表の構成法に依存して異なったものになろう．

記号表のエントリの属性に関しては，整数値をとるつぎのような5通りの要素だけを考慮しよう．

1. TYPE. 型の指定. 0 = UNDEFINED, 1 = REAL, 2 = INTEGER, 3 = BOOLEAN, 4 = LABEL.
2. TYPE1. 1 = 単純変数, 2 = 配列名, 3 = 添字付変数.
3. TEMPORARY. 1 = 中間結果用変数, 0 = それ以外.
4. DECLARED. 0 = 未宣言, 1 = 宣言済.
5. ADDRESS. 名札の4-組番号，または記号表の別のエントリの番地.
6. NUMBER. 配列の場合，次元の数.

UNDEFINED, REAL, INTEGER, BOOLEAN, LABEL は，われわれのコンパイラでは，上述した定数値を値に持つ整数型変数と考えていることを注意しておくべきであろう．被演算子に関する記述は全て，同じ表に貯えられており，被演算子の内部表現は，被演算子を記述している記号表の要素へのポインタになっている可能性がある．これによって，表記が簡単になり，われわれの注意を他の問題に集中させることができる．記号表のエントリへのポインタPが与

えられると，P. TYPE, P. TYPE1 などによってその属性を参照することができる.

名札が定義されると，その記号表のエントリのADDRESS成分には，その名札が指している4-組の番号が入る.

添字付変数は，TYPE1 = 3 となっているエントリによって表される．この場合，TYPEは，配列要素の型であり，NUMBERにはこの配列を記述している記号表のエントリの番地が入り，ADDRESS は，この添字付変数の VARPART を記述しているエントリの番地が入る.

意味用スタックの参照

問題にしている文形式の把手 x が，（規則 $U::=x$ を用いて）非終端記号 Uに還元されようとするとき，記号に意味付けするルーチンが呼ばれる．このルーチンには,

1. 記号表に情報を納めたり，調べたりする.
2. x に関連する意味情報を検査する.
3. 4-組を生成する.
4. 非終端記号Uと意味情報を結合する.

という仕事が含まれている．第12章と同様に，記号 U に結び付いた情報 S を U. S で参照しよう．例えば，記号〈式〉は，式を記述している記号表のエントリの番地を値として持つポインタ ENTRY と結び付けられていた．これは,

〈式〉. ENTRY

で参照される.

どんなコンパイラでも，〈式〉は，構文用スタックのどこかにはのっていることを思い出してほしい．一方〈式〉. ENTRY の値は，意味用スタックのそれに対応する場所にあるだろう．以下に，これから使おうとしている記号の主なものを，その意味とともに列挙しておこう.

1. I. NAME　NAME は名前を形成する実際の文字列を指す.
2. 〈変数〉. ENTRY　ENTRY には記号表中の，当該の変数のエントリへのポインタが入る.
3. 〈式〉. ENTRY　〈変数〉の場合と同様.
4. 〈条件節〉. JUMP　〈条件節〉に対する規則の右辺は IF〈式〉THEN で

ある．構文解析を行うと，BZ 4-組が生成されるが，まだ分岐先は未知である．したがって，JUMP には，後に，分岐先番地がわかったときに代入するために BZ 4-組の番号を入れておく．

使用されるサブルーチンと変数

意味付けルーチンは，ALGOL で書くが，ここで使用するサブルーチンと変数は以下のとおりである：

1. ENTER(＋, P1, P2, P3)．4個の引数を成分とする4-組を，原始プログラムの内部表現の一覧表に加える．
2. ERROR(i)．エラーメッセージ i を印刷して，翻訳を中止するルーチン．
3. NEXTQUAD．つぎに生成される4-組の番号を保持している整数型変数．初期値は1である．
4. QUAD(I, J)．これまでに生成されたI番目の4-組のJ番目の成分を参照する．これを使用するのは，この成分を参照したり書換えたりするときである．
5. GENERATETEMP(P)．中間結果用変数に対して記号表のエントリを開設し，そのエントリの番地をポインタ変数Pにのせるルーチン．
6. CHECKTYPE(P, N)．ポインタPの指す記号表のエントリを調べて，その型がNであるかどうかを検査する（例：CHECKTYPE(P, BOOLEAN) など）．もし型が一致しなければ，エラーメッセージを印刷して，翻訳を中止する．
7. CONVERTRI(P)．エントリPが INTEGER であるかどうかを調べて，もし INTEGER なら，そのまま戻る．もし REAL なら，GENERATETEMP を呼出して，新しい中間結果用変数（INTEGER 型）を得る．CVRI 4-組を生成して，入力の値を整数型に変換し，それを新しい中間結果用変数にしまう．最後に新しい中間結果用変数の記号表におけるエントリの番地を引数Pに置換える．もし，エントリPの型が INTEGER でも REAL でもなければ，エラーメッセージを印刷する．
8. CONVERTIR(P)．これは，INTEGER から REAL への変換．
9. コンパイル時に必要に応じて作業用変数（例，P, P1, P2 など）が用いられる．これらの変数の型と用法は，文脈から明らかであろう．

13.2 条　件　文

まず，条件文用のルーチン群から始めよう．これらは，簡単ではあるが，いろいろな問題点を含んでいるからである．条件文の普通の定義は

〈実行文¹〉::＝〈条件節〉〈実行文²〉ELSE〈実行文³〉
　　　　　｜〈条件節〉〈実行文²〉
〈条件節〉::＝IF〈式〉THEN

である．ここで〈式〉はBOOLEAN型でなければならない．0であれば偽，0でなければ真と仮定し，式の値は評価されて，中間結果用変数Tに代入されるものとすると，つぎのような4-組の系列を生成することになろう：

(1)	T:＝〈式〉に対応する4-組		(1)	T:＝〈式〉に対応する4-組
(p)	BZ $q+1$, T, 0	または	(p)	BZ q, T, 0
	〈実行文²〉			〈実行文²〉
(q)	BR r		(q)	
($q+1$)	〈実行文³〉			
(r)				

BZ 4-組は，〈条件節〉::＝IF〈式〉THENに対応するルーチンによって生成される．もちろん，分岐先は未知のままである．そこで〈条件節〉に連らなる意味として後で使うために4-組の番号をとっておく．〈式〉を評価する4-組の系列は，それがどんなに複雑なものであっても生成されていると仮定できることに注意しよう．また，〈式〉は，この系列と，記号表の中でこれを記述しているエントリの番地とをENTRYの中で結び付けている．

〈条件節〉::=IF〈式〉THEN

P:=〈式〉.ENTRY; CHECKTYPE(P, BOOLEAN); 〈条件節〉.JUMP:=NEXTQUAD;	Pに，記号表の〈式〉に対するエントリへのポインタを代入．つぎの4-組の番地を憶えておく．これはTHENにつづく実行文を飛越える分岐BZである．
ENTER(BZ, 0, P, 0);	BZ 4-組の生成．

CHECKTYPEの呼出しは，式がBOOLEAN型であることを確かめるものであることに注意．

このルーチンは，偽の場合の分岐のためのコードを生成することを考慮して

いる．さて ELSE 以下がない場合の条件文に対するルーチンを示そう．〈条件節〉〈実行文^2〉が〈実行文^1〉に還元されることになった時には，〈実行文^2〉に対する4-組の系列はすでに生成されていると仮定できる．もちろん，IF〈式〉THEN が〈条件節〉に還元されてしまうまでは，〈実行文^2〉は，全く構文解析されないので，〈実行文^2〉に対する 4-組の系列は，BZ の後に位置している．この場合，われわれが行わなければならないことは，〈実行文^2〉を飛越えてゆく BZ を埋込むことしかない．この4-組の番号を〈条件節〉. JUMP にとっておいたことを想い出してほしい．したがって，意味付けルーチンは，

〈実行文^1〉::=〈条件節〉〈実行文^2〉

I:=〈条件節〉. JUMP; QUAD(I, 2):=NEXTQUAD;	つぎの 4-組の番号を BZ 4-組(I 番目の 4-組)の第 2 成分に置く．

上の規則を用いて ELSE 以下を持つ条件文のコードを生成するときは，問題が生ずる．すなわち，〈実行文^2〉と〈実行文^3〉の間に分岐命令を置かなければならない．しかし，上の構文を用いれば，〈実行文^2〉も〈実行文^3〉もともに，こうなる前に構文解析が済んでいる．われわれが望んでいるような意味に構文を合わせるために，構文を

〈実行文^1〉::＝〈真の部〉〈実行文^3〉
〈真の部〉::＝〈条件節〉〈実行文^2〉ELSE

と変更しよう．これによって，以下の二つの意味付けルーチンを書くことができる．

〈真の部〉::=〈条件節〉〈実行文^2〉ELSE

〈真の部〉. JUMP:=NEXTQUAD;	つぎの 4-組の番地を憶えておき，後に分岐先をここに埋込む．
ENTER(BR, 0, 0, 0) I:=〈条件節〉. JUMP; QUAD(I, 2):=NEXTQUAD;	〈実行文^2〉を飛越える BZ の第 2 成分を決定する．

〈実行文^1〉::=〈真の部〉〈実行文^3〉

I:=〈真の部〉. JUMP; QUAD(I, 2):=NEXTQUAD	〈実行文^3〉を飛越える分岐命令の番地を埋込む．

繰返すが，この意味付けルーチンは，基本的な 2 点を説明している．第 1 点

は，把手 $XY\cdots Z$ が，還元されようとするときは，この把手に含まれる非終端記号に対するコードは全て生成が済んでいること，もしこのような非終端記号同士の間に特別にコードを挿入しなければならない事情があれば，意味に合うように構文を変更しなければならない．

第2点は，後で必要になる情報をとっておくために，記号に結び付けられた意味を用いることである．条件文は，何重にも入れ子になって構文解析のある一時点でいくつもの条件節を作り出すことがありうることに注意しよう．それぞれが，対応する JUMP 成分（BZ の番地部）を持っており，構文の機構——スタック——によって，それらは自動的に分離されて処理される．

今述べた2点は，以後にも繰返し繰返しお目に掛かるだろう．

演習問題　13.2

1. IF A ≤ B THEN 〈実行文^2〉のような実行文に対して生成されたコードはかなりまずいものである．まずAからBを引いて，結果が正か否かによって真か偽を決め，〈実行文^2〉を BZ する（訳注，偽ならば飛越える）．〈条件節〉の構文を，IF 〈式^1〉≤〈式^2〉THEN という特別な場合も認識するように書換え，この場合には（BG, 0, 〈式^1〉, 〈式^2〉）という単一の4-組を生成するようにせよ．

2. IF〈式^1〉≥〈式^2〉を認識するときも，構文と意味を変更して（BG, 0, 〈式^2〉, 〈式^1〉）を生成するようにせよ．

13.3　名札と分岐

名札の定義は，構文によって

〈実行文^1〉∷＝I:〈実行文^2〉

である．ここで "I" は名前を示す非終端記号である．ここでも条件文の場合と同様な問題点が存在する．つまり名札 I には，〈実行文^2〉の先頭番地が与えられなければならないのに，それが不可能だからである．何となれば，最終的にこの名前を処理しようという時点では，〈実行文^2〉に対するコードがすでに生成済みとなっているからである．そこで，上の構文を変更して

〈実行文^1〉∷＝〈名札の定義〉〈実行文^2〉

〈名札の定義〉∷＝I:

とする．こうすれば，名札は，実際に定義される前に参照できて，以下のような意味付けルーチンが書ける．

〈名札の定義〉::=I:

LOOKUPDEC(I.NAME, P);	記号表で名前を引く．
IF P = 0 THEN	
BEGIN	
INSERT(I.NAME,P);	名前が記号表になければ，
P.TYPE:=LABEL	それを記号表に加えて，名札であるしる
END	しをつける．
ELSE BEGIN CHECKTYPE(P, LABEL);	記号表にあれば，それが名札かどうか調
IF P.DECLARED = 1	べる．宣言済みビットが1であれば二重
THEN ERROR(3)	定義なのでエラー，メッセージ3を印刷
END;	する．
P.DECLARED:=1;	宣言済みビットを設定し，番地を定義す
P.ADDRESS:=NEXTQUAD	る．

〈実行文^1〉::＝〈名札の定義〉〈実行文^2〉

に対しては，これ以上意味を追加する必要はないだろう．

FORTRAN では，GO TO 文は問題が生じない．それは，文の番号が，一意的でなければならないからである．GO TO I に対する意味付けルーチンはつぎのとおり：

〈実行文〉::=GO TO I

LOOKUP(I.NAME, P);	記号表で名前Iを探す．
IF P = 0 THEN	
BEGIN	記号表になければ，その名前を記号表にのせ
INSERT(I.NAME, P);	る．
P.TYPE:=LABEL;	型はLABEL.
P.DECLARED:=0;	未宣言である．
END	
ELSE CHECKTYPE(P, LABEL);	記号表にあり，型を検査して，分岐命令を生
ENTER(BRL, P, 0, 0);	成する．

分岐命令の連結

名札への分岐には，BRL 命令の代りに，BR 命令を使用する方が望ましいかもしれない．すなわち，命令の被演算子欄に，記号表のエントリの番地を置く代りに，4-組の番号を直接置きたいのである．このような問題は，分岐に対して，機械命令を生成するときにも生じ，分岐先の命令の番地を，分岐命令の番

地部に置く.

しかしながら GO TO L は, L が定義される前に現れるかもしれないので, 名札の定義が構文解析されるまでは, 4-組の番号を埋込むわけにはゆかない. 同じ名札に分岐する同じような GO TO 文がいくつかあり得るので, 問題は複雑になる.

参照を前向きに進めてゆく簡単な方法は GO TO L という文を 1 次元リストに結びつけて, このリストの先端を, 記号表の L の ADDRESS 成分に置いておく. 図 13.1 に, 10番, 20番, 40番の 4-組が GO TO L である例を示した. この 1 次元リストの結合情報をとっておくための, 特別な表やスタックは必要ない. このためには, 4-組の被演算子の第 1 欄を使えばよい.

```
GO TO L      (10) BR 0, 0, 0
  :               :
GO TO L      (20) BR 10, 0, 0
  :               :
GO TO L      (40) BR 20, 0, 0
```

NAME	ADDRESS	DECLARED
L	40	0

原始プログラム　　4-組　　記号表中の各欄

図 13.1　前向き参照のための結合状況

L の定義が最終的に構文解析されると, 意味付けルーチンが, この 1 次元リストをたどって, 全ての分岐命令に正しい4-組番号を挿入する. このような機構は, 前向きに参照をすすめる場合の非常に簡単な解決法であると同時に, 能率もよい. このやり方は, 原始言語によっては, 変数や手続きを前向きに参照してゆく場合にも時々使われることがある.

ブロック構造がある場合の前向き参照

ブロック構造を持った言語に前向き参照を行うと問題が出てくる. 名札が外側のブロックでは定義済みであるが, その名札を参照しているブロック内では未定義の場合, この名札の定義が, このブロック内で行われるかどうかという問題である.

```
BEGIN L:BEGIN GO TO L
```

の最後の文を構文解析するところで, GO TO L が, 外側の L を参照しているのか, まだ構文解析されていない名札の定義を参照しているのかという問題で

ある．解決法の一つは，第1パスではブロック構造と宣言だけを見て，第2パスで，参照している名札がどこで定義されているかを調べることである．1-パスコンパイラが望ましい場合には，実際に名札を定義しているブロックで名札定義宣言——例えば LABEL L——を要求する解決法もある．これはプログラムする側から見ればとんでもない解決法で滅多には持ちだせない．名札の宣言は，プログラムをする手間を不必要に増加させてしまう．もう一つの解決法は，コンパイラ書きの側からは，多少面倒が増えるもので，以下で簡単に概説しよう．

ブロックに入ってそのブロック中で未定義な名札が参照されると，そのブロックの名札として記号表にエントリを開設する．ブロックを開始するごとに，またブロック中で上のような名札に会うごとに，先程述べたように，この名札を参照する4-組の1次元リストに結合する．このブロックの中で，実際に名札が定義されれば，その時全てうまく運ぶ．1次元リスト上の4-組は正しい名札を参照できる．もしこのブロック中で当の名札が定義されなければ（これがわかるのは，ブロック全体が構文解析された時点である），この名札のエントリを記号表から取除き，（すぐ外側のブロックでこの名札がまだ使用されていなければ）この名札を外側のブロックの記号表にのせる．そして，今閉ざされたブロックでの参照の1次元リストを，すぐ外側の同じ名札の参照の1次元リストに結合する．

もちろん，コンパイラの作り方によってはブロック脱出用ルーチンの呼出しを要するものもあるのでこの場合には，ブロックからの脱出の問題がまだ残っている．L への分岐に必要なブロック脱出命令も一しょにしておいて，L のブロック番号が最終的に決定したときこの命令の中に，正しいパラメタを埋込んでやればよい．

演習問題　13.3

1. 分岐と名札に与えた意味付けルーチンを変更して，BRL 命令の代りに BR 命令を生成するようにせよ．
2. ブロック構造を持った言語において前向き参照を解決する機構を可能にするような意味付けルーチンを設計し，作成せよ．

13.4 変数と式

ここで対象とする ALGOL 型の規則は

〈変数〉:: = I|I (〈式の並び〉)

〈式の並び〉:: =〈式〉|〈式の並び〉,〈式〉

である．まず，〈変数〉:: = I では，4-組を生成する必要はない．ここでは，記号表で名前を探索してそのエントリを〈変数〉と結合しなければならない．したがって，意味付けルーチンはつぎのようになる．

〈変数〉::=I

LOOKUP(I. NAME, P); IF P = 0 THEN ERROR(4); 〈変数〉. ENTRY:=P;	記号表で名前 I を引く．もし P = 0 ならば，表にないのでエラーメッセージの印刷．記号表のエントリを〈変数〉と関連づける．

もし FORTRAN のコンパイルを行っているのならば，名前が表に見付からなくとも誤りではない．この場合は，手続き INSERT を用いて，記号表にこの名前をのせ，この名前の最初の英字に応じて整数型か実数型かその属性を決めてやる．

添字付変数 I(〈式の並び〉) を使用すると，コンパイラ側の手間がふえる．実際，全ての添字を評価する 4-組を生成し，8.4 節に述べた機構を用いて参照される配列要素の番地を計算する 4-組を生成しなければならない．さらに，添字の個数を数え，宣言されている個数と合致していることを確かめたい．

上に与えた構文を使用することもできようが，VARPART の計算に (8.4.5) を用いるためにはこれを変更して

〈添字等〉 :: = I (〈式〉|〈添字等〉,〈式〉

〈変数〉 :: =〈添字等〉)

とする．配列名用の記号表は TYPE1 = 2 となっており，TYPE 欄は各配列要素の型，NUMBER 欄は次元数である．問題を簡単にするために記号表には引き続く番地に，d_1, d_2, $\cdots$, d_n をのせる (8.4.5参照)．これは記憶場所の無駄遣いではあるが，実際の段取りを説明するのには最もたやすい．この取扱いは普通は，ドープ・ベクトルの実行時の番地を ADDRESS 欄に置き，この番地を

用いて $d_1, d_2, \cdots$ などがドープ・ベクトルのどこにあるかを知るようになっている．

さて〈添字等〉::=I (〈式〉という規則に対する意味付けルーチンに戻ろう．ここで行わねばならないのは，表で配列名を引き VARPART:=〈式〉というコードを生成し，d_1 に対する記号表のエントリの番地と〈添字等〉を結び付けることである．

〈添字等〉::=I (〈式〉

LOOKUP (I. NAME, P);	名前Iを探索する．
IF P=0 OR P. TYPE ≠ 2 THEN ERROR (5);	宣言済みであり，型はARRAYでなければならない．
〈添字等〉. COUNT := P. NUMBER−1;	ここでは第1添字だけしかない．COUNTは，次元の数を検査するために用いる．
〈添字等〉. ARR:=P;	配列名のエントリを憶えておく．
〈添字等〉. ENTRY:=P+1;	d1のエントリをENTRYに憶えておく．
GENERATETEMP (P); P. TYPE:=INTEGER;	VARPARTのために INTEGER 型の中間結果用変数を生成．
〈添字等〉. ENTRY 2:=P;	そのエントリの番地を記憶する．
CONVERTRI (〈式〉. ENTRY);	このルーチンはエントリを調べ必要ならば変換を行う．
P:=〈式〉. ENTRY; ENTER (:=, P, ,〈添字等〉. ENTRY 2);	第1変数をVARPARTに代入する4-組を生成する．

〈添字等〉に関連する意味情報を列挙しておこう．

1. 〈添字等〉.ENTRY　　もし i 番目の添字のコードが処理されていれば，d_i に対する記号表のエントリの番地．
2. 〈添字等〉.ENTRY2　　VARPART に対する記号表のエントリの番地．
3. 〈添字等〉.COUNT　　i 番目の添字が生成されていれば，(次元数) $-i$.
4. 〈添字等〉.ARR　　配列名表のエントリへのポインタ．

このつぎの段階は，〈添字等¹〉::=〈添字等²〉，〈式〉に対するルーチンのプログラムである．これが，添字 i であると仮定すると，VARPART:=VARPART $* d_i +$〈式〉に対するコードを生成しなければならない．

〈添字等^1〉:=〈添字等^2〉，〈式〉

〈添字等^1〉. COUNT 　:=〈添字等^2〉. COUNT−1;	添字の数を数える.
〈添字等^1〉. ARR:=〈添字等^2〉. ARR;	配列要素の型の記憶.
〈添字等^1〉. ENTRY 　:=〈添字等^2〉. ENTRY+1;	〈添字等^1〉. ENTRY は d_i 用の表のエントリへのポインタとなる.
P1:=〈添字等^2〉. ENTRY 2; 〈添字等^1〉. ENTRY 2:=P1;	記号表中の VARPART のエントリを P1 と ENTRY 2 に代入.
ENTER(*, P1, 〈添字等^1〉. ENTRY, P1);	VARPART:=VARPART*d_i に対する4-組.
P:=〈式〉. ENTRY; CONVERTRI(P);	添字を（もちろん必要な場合だけ）整数型に変換する4-組の生成.
ENTER(+, P1, P, P1);	VARPART:=VARPART+添字 用の4-組

添字式がどんなに複雑であっても問題ではないことに注意しよう．添字式がまた別の配列や関数などを参照していたりすることもありうる．そのときでもその計算結果を記述している記号表のエントリへのポインタを見るだけである．最後に〈変数〉::=〈添字等〉)のためのルーチンをプログラムしなければならない．このルーチンで，添字の個数を検査し，配列要素を記述している表のエントリを組立てることができる．

〈変数〉::=〈添字等〉)

IF 〈添字等〉. COUNT ≠ 0 THEN ERROR(6);	COUNT=0 はここでは添字の個数が正しいことを示す.
GENERATETEMP(P); P. TYPE 1:=3;	配列要素を記述するための中間結果用変数を生成する.
P. TYPE:=〈添字等〉. ARR. TYPE;	型を埋める.
P. ADDRESS:=〈添字等〉. ENTRY 2;	VARPART のエントリの番地を埋める.
P. NUMBER:=〈添字等〉. ARR;	配列名表のエントリの番地を埋める.

算術演算子と論理演算子

2 項演算子は12章に述べたと同じように取扱う．ここでは実際これ以上詳細に渉る必要はない．〈式^1〉::=〈式^2〉+〈項〉のような規則に対しては，〈式^2〉と〈項〉は型の一致を検査して，必要に応じて，どちらかの被演算子の型を新しいものに変換する4-組が生成される．結果を代入するための中間結果用変数が新しく生成され，演算子に対応する4-組が生成される．単項演算子も同様に

取扱う.

演習問題 13.4

1. 規則〈式^1〉::＝〈式^2〉＋〈項〉に対する意味付けルーチンを書け.
2. 規則〈因子〉::＝－〈1次子〉に対する意味付けルーチンを書け.

13.5 FOR ループ

```
〈実行文〉::＝FOR〈変数〉:＝〈繰返し要素の並び〉DO〈実行文〉
〈繰返し要素の並び〉::＝〈式¹〉STEP〈式²〉UNTIL〈式³〉
```

によって定義される ALGOL の FOR ループに対する意味付けルーチンを説明しよう. ここで, 〈式^2〉の値はつねに正であると仮定しておこう. この仮定のもとに, FOR ループは

```
〈変数〉:＝〈式¹〉;
GO TO OVER;
AGAIN:〈変数〉:＝〈変数〉＋〈式²〉;
OVER:
IF〈変数〉≤〈式³〉†
THEN BEGIN〈実行文〉; GO TO AGAIN END
```

と同等である.

ループの制御変数が各所で使われていることに注意. したがって, このループを構文解析しながら, 記号表のこの変数のエントリへのポインタを都合のよい場所にとっておかねばならない. 4-組を正しい順序で生成するためには, 規則をいくらか変更しなければならないことが, これまでの議論で, かなり明らかになっただろう. ここでは

```
〈for 1〉::＝FOR〈変数〉:＝〈式¹〉
〈for 2〉::＝〈for 1〉STEP〈式²〉
〈for 3〉::＝〈for 2〉UNTIL〈式³〉
〈実行文¹〉::＝〈for 3〉DO〈実行文²〉
```

を用いる. 以下のような意味付けルーチン群を書くことができる. これらのル

† 訳注 〈式^2〉が正と仮定しているのでこの不等号でよい.

ーチンでは，被演算子の型を検査して，型変換を行う4-組を生成することはしていない．筋道を追い易くするためである．

〈for 1〉::=FOR〈変数〉:=〈式^1〉

LOOKUP (〈変数〉. NAME, P);	記号表中の番地をPに.
IF P = 0 THEN ERROR (7);	
P1:=〈式^1〉. ENTRY;	ループの制御変数を初期設定するコードの生成.
ENTER (:=, P1, , P);	
〈for 1〉. ENTRY:=P;	ループ制御変数のエントリの番地とOVERにBRするコードの番地を記憶しておく.
〈for 1〉. JUMP:=NEXTQUAD;	
ENTER (BR, 0, 0, 0);	OVERにBRする4-組の生成.
〈for 1〉. JUMP2:=NEXTQUAD;	AGAINと名付けられた4-組の番地も憶えておく必要がある.

〈for 2〉::=〈for 1〉 STEP 〈式^2〉

〈for 2〉. JUMP2:=〈for 1〉. JUMP2;	名札AGAINの番地とループ制御変数のエントリの番地を退避する.
〈for 2〉. ENTRY:=〈for 1〉. ENTRY;	
P:=〈式^2〉. ENTRY;	
ENTER (+, 〈for 1〉. ENTRY, P, 〈for 1〉. ENTRY);	増分の値を制御変数に加えるための4-組を生成する.
I:=〈for 1〉. JUMP;	OVERを定義して，先に未定義だった分岐先を埋込む.
QUAD (I, 2):=NEXTQUAD;	

〈for 3〉::=〈for 2〉 UNTIL 〈式^3〉

〈for 3〉. JUMP2:=〈for 2〉. JUMP2;	AGAINの番地の退避.
〈for 3〉. JUMP:=NEXTQUAD;	IFに対する条件付分岐を生成しようとしているのでその番地を憶えておかねばならない.
ENTER (BG, 0, 〈for 2〉. ENTRY, 〈式^3〉. ENTRY);	これは IF〈名前〉≦〈式^3〉に対するコードである.

〈実行文^1〉::=〈for 3〉 DO 〈実行文^2〉

ENTER (BR, 〈for 3〉. JUMP2, 0, 0);	これはGO TO AGAINである. AGAINの4-組番号はJUMP2にとっておいた.
I:=〈for 3〉. JUMP;	BG, 〈名前〉, 〈式〉 の番地部がこれで埋められる.
QUAD (I, 2):=NEXTQUAD;	

演習問題　13.5

1. 上で議論したと同じループについて，意味付けルーチンを書け．ただし，増分要素は必ずしも正とは限らないことにする．

2. 増分要素と終値要素〈式^3〉は一度だけしか評価されないものとして，同じ for ループについて，意味付けルーチンを書け.

3. つぎのような構文をもった WHILE 文
〈実行文〉::＝WHILE〈式〉DO〈実行文〉
に対する4-組の系列を生成する意味付けルーチンを書け.

4. 本節で述べたループに対して，制御変数が，整数であり，全ての〈式〉が定数の場合に，もっと能率のよいコードを生成するように意味付けルーチンを書換えよ. つぎのような4-組（TEST,〈変数〉,〈定数^1〉,〈定数^2〉）を仮定せよ. これは，(1)〈定数^1〉を〈変数〉に加え，(2) その結果が〈定数^2〉より大であればつぎの4-組を実行し，さもなければ，つぎの4-組をスキップせよ. という意味である.

13.6 論理式の最適化

この節では，つぎの構文

(13.6.1) Z::＝E
E::＝T | E OR T
T::＝F | T AND F
F::＝I | (E) | NOT F

で定義される論理式のコードを生成する方法を2通り示したい. ここでの名前は，論理型で，真と偽の値をとる. 3個の演算子 OR, AND, NOT は下記の表で定義する：

OR	真	偽
真	真	真
偽	真	偽

AND	真	偽
真	真	偽
偽	偽	偽

NOT	真	偽
	偽	真

構文から AND は OR より順位が高く，NOT が最も高い順位を持っていることがわかる. NOT NOT A は A と同等であることに注意. このような論理式を評価する方法は算術式の評価と同じで，演算子を左から右に評価して，それらの順位とかっこの有無を考慮して行うのが普通である. A AND (B OR NOT C) に対するポーランド記法は A B C NOT OR AND となろう. このように

挿入記法(インフィクス)から後置記法(ポストフィクス)（あるいは4-組）に変換する意味付けルーチンのプログラムは，容易に書けるだろう．しかしながら，論理式の評価には，もっと能率のよい方法がある．

論理式の最適な評価

論理式 A AND(B OR NOT C) を考えよう．もし A が偽なら，これ以上評価をつづける必要はない．結果は偽である．同様に，もし A が真で B が真ならば，結果は真である．NOT C を評価する必要はない．われわれは，左から右に評価を行って，結果がわかったところで直ちに論理式の評価を中止したい．ところで，上にあげた式は

```
IF A
THEN IF B
     THEN TRUE ELSE NOT C
ELSE FALSE
```

と等価であり，これがわれわれが評価したい方法である．論理式の構文と意味を改めて

(13.6.2)　Z :: = E
　　　　E :: = T | T OR E
　　　　T :: = F | F AND T
　　　　F :: = I | (E) | NOT F

と定義し，ここで

(13.6.3)　c OR d は IF c THEN TRUE ELSE d
　　　　c AND d は IF c THEN d ELSE FALSE
　　　　NOT c は IF c THEN FALSE ELSE TRUE

とする．われわれは，名前，定数 0, 1 と 2 個の4-組（: =, I, , T）と（B, I, i, j）を用いて 4-組を生成したい．ただし I は名前である．後の方の4-組は，つぎのような意味である．もし I が真(1)であれば，4-組 i に分岐，さもなければ，j に分岐する．

図 13.2 に例をいくつか示す．これから，二，三の事実に気付いてほしい．まず，結果はつねに変数 X に代入されている．第 1 番目の4-組は 1 を X に代入する．つまり論理式はまず真と仮定している．もし真ならば，最後の行のつぎの

4-組に分岐する．もし偽ならば，最後の行に分岐してXに0を代入する．

つぎに，4-組に現れる名前の順番は，もとの論理式に現れる順番と同じである．最後に，NOT に対しては，演算子は現れない．もし A が (B, A, *i*, *j*) と表されていれば，NOT A は (B, A, *j*, *i*) で表せる．ただ，真と偽とへの分岐先を入換えるにすぎない．

一たん，この節の内容を理解すれば，同じようにポーランド記号列を生成し，あるいは機械語すら生成できることがはっきりするだろう．そして1-パスコンパイラでも同じことである．もちろん，いろいろ細かい点はあるが，方法は同じである．

われわれの興味の中心は，分岐先の番地がどのようにして間違いなく埋込まれるかを確かめることを示すことにある．つまりこれが問題の核心である．したがって，ここでは，型の一致とか，記号表上の名前のエントリなどには，気を配らないことにしよう．

(1) (:=, 1, , X) (2) (B, A, 5, 3) (3) (B, B, 5, 4) (4) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 3, 5) (3) (B, B, 5, 4) (4) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 3, 4) (3) (B, B, 5, 4) (4) (:=, 0, , X)
A OR B	NOT A OR B	A AND B
(1) (:=, 1, , X) (2) (B, A, 6, 3) (3) (B, B, 6, 4) (4) (B, C, 5, 6) (5) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 6, 3) (3) (B, B, 4, 6) (4) (B, C, 6, 5) (5) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 3, 5) (3) (B, B, 6, 4) (4) (B, C, 6, 5) (5) (:=, 0, , X)
A OR B OR NOT C	A OR NOT B OR C	A AND (B OR C)
(1) (:=, 1, , X) (2) (B, A, 4, 3) (3) (B, B, 6, 4) (4) (B, C, 5, 7) (5) (B, D, 7, 6) (6) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 3, 4) (3) (B, B, 4, 7) (4) (B, C, 6, 5) (5) (B, D, 7, 6) (6) (:=, 0, , X)	(1) (:=, 1, , X) (2) (B, A, 3, 6) (3) (B, B, 5, 4) (4) (B, C, 7, 5) (5) (B, D, 7, 6) (6) (:=, 0, , X)
(A OR NOT B) AND (NOT C OR D)	(A AND NOT B) OR (NOT C AND D)	(A AND (NOT (B OR NOT C) OR D))

図 13.2 4-組による論理式の表現

13.6.1 上向き解析法

生成規則の並べ換え

文法 (13.6.2) を用いると，例えばE :: = T OR Eに対するルーチンが呼ばれた時点では，T に対する4-組も E に対する 4-組も生成済みとなっている．しかし，ここで生成したいコードは

(1)　(B, T, 3, 2)
(2)　(E に対する4-組)

のようなものである．すなわち，もしTが真であれば，Eの評価はしないで飛び越してしまいたい．このためには，与えられた規則を少々分解してTとEが同一の右辺に現れないようにしなければならない．文法をつぎのように変更する．

```
(13.6.3)  Z     :: = E
          E     :: = T | TOR E
          TOR   :: = T OR
          T     :: = F | FAND T
          FAND  :: = F AND
          F     :: = I | (E) | NOT F
```

意味情報

論理式の構文解析の最中，把手が（例えば）名前Aであるときはいつでも，(B, A, 0, 0) という4-組を生成する．Aが真 (1) のとき，偽（0）のときそれぞれの分岐先の番地が未知である．したがって，われわれは真のときの分岐先を埋込まねばならぬことを憶えておくため TCHAIN と，F（規則は F :: = I）とを，同様に偽のときの分岐先を埋込まねばならぬことを憶えておくために FCHAIN と F とを結び付ける．番号 k が 4-組の第3成分と第4成分のどちらを参照しているかを区別するために，第3成分の場合は k，第4成分の場合は $-k$ のようにして使用する．したがって，論理式の最初の名前が構文解析された後には，F. TCHAIN = 2 と F. FCHAIN = − 2 で，4-組は

(1)　(: = , 1, , T)
(2)　(B, A, 0, 0)

となっている．

任意の非終端記号 E, TOR, T, FAND, あるいは F を考えよう．いずれも，いくつかの名前を含んだかなり複雑な論理式を表わすことができる．したがって，いずれも，真あるいは偽のときどこに分岐すればよいか，未知のまま一連の長い分岐のリストになる可能性がある．そこで，真の場合の分岐を全て連ねて，最初のものを非終端記号の TCHAIN に憶えておく．同様に偽の場合については FCHAIN に憶えておく．このように13.3節に述べたと同じようにして，まさしく前向き参照の1次元リストを作り上げていく．

例えば，部分論理式 (A OR B AND C) を考えよう．すでにこれを F まで還元したと仮定する．4-組の系列は

(1) (:=, 1, , T)
(2) (B, A, 0, 3)
(3) (B, B, 4, 0)
(4) (B, C, 0, 0)

のようになっているはずである．ただし，0は，分岐先が未だ不明であることを示している．しかしながら，もし0となっている欄を連結するならば，

F. TCHAIN = 4, F. FCHAIN = −4

で

(1) (:=, 1, , T)
(2) (B, A, 0, 3)
(3) (B, B, 4, 0)
(4) (B, C, 2, −3)

となるだろう．下記のような2個の手続 FILLIN と MERGE を意味付けルーチンの中で用いて，上のような1次元リストを取扱う補助とする．意味付けルーチン自身は，4-組の系列の最初が (:=, 1, , T) となるように初期設定されていると仮定している．これらのルーチンを初めて見るときは，まず非終端記号Fに対するものから始めて，自分の方向へ進む．こうすると処理過程の理解が一層容易になる．

PROCEDURE FILLIN(K); VALUE K; INTEGER K; BEGIN INTEGER SAVE; WHILE K ≠ 0 DO BEGIN I:=IF K>0 THEN 3 ELSE 4; SAVE:=ABS(K); K:=QUAD(SAVE, I); QUAD(SAVE, I):=NEXTQUAD END END	FILLIN は，つぎの 4-組の番号を，K から始るリストに埋込む． K が 0 であればリストは尽きた． K の正負によって，第 3 欄か第 4 欄かを決める． K で示している 4-組番号を SAVE に． 1 次元リスト上のつぎの要素を K に． つぎの 4-組の番号をその欄に埋込む．

PROCEDURE MERGE(K1, K2); INTEGER K1, K2; BEGIN INTEGER K; IF K1 = 0 THEN K1:=K2 ELSE BEGIN K:=K1; LOOP: I:=IF K>0 THEN 3 ELSE 4; K:=ABS(K); IF QUAD(K, I) ≠ 0 THEN BEGIN K:=QUAD(K, I); GO TO LOOP; END; QUAD(K,I):=K2; END; END	K1 と K2 は，それぞれ 1 次元リストの最初の要素の番号が入っている変数． リスト K2 をリスト K1 の後に連結する．もし K1 = 0 なら，K2 をそのまま K1 に． リスト K1 をたどって，最後の要素を K で指す． リスト K1 の最後の要素に K2 を置く．

Z ::=E

FILLIN(E. FCHAIN); ENTER(:=, 0,, X); FILLIN(E. TCHAIN);	処理は全部終了．偽の場合の分岐先を全て埋込み，最後の 4-組を生成． 真の場合の分岐先を全て埋込む．

E ::=T

E.TCHAIN:=T. TCHAIN; E.FCHAIN:=T. FCHAIN	E と T は意味上では同じもの．したがって，真，偽両リストの端を転写する．

E^1 ::=TOR E^2

E^1. FCHAIN:= E^2. FCHAIN; E^1. TCHAIN:=TOR. TCHAIN; MERGE(E^1. TCHAIN, E^2. TCHAIN)	TOR の偽となる分岐先はすでに埋込まれているので E^2 の場合だけ転写. 真の場合の 2 通りの分岐先のリストを 1本のリストに連結.

TOR ::=T OR

FILLIN(T. FCHAIN); TOR. FCHAIN:=0; TOR. TCHAIN:=T. TCHAIN	T において偽の場合の分岐を全て，ポーランド記法でのつぎの記号に行くように埋める.真の場合の分岐先のリストを退避する.

FAND ::=F AND

FILLIN(F. TCHAIN); FAND. TCHAIN:=0; FAND. FCHAIN:=F. FCHAIN	F が真となる場合の分岐先を全てポーランド記法のつぎの記号に行くように埋込む. 偽の場合の分岐先は，未知.

T ::-F

T. TCHAIN:=F. TCHAIN; T. FCHAIN:=F. FCHAIN	説明は不要.

T^1:=FAND T^2

T^1. TCHAIN:= T^2. TCHAIN; T^1. FCHAIN:=FAND. FCHAIN; MERGE(T^1. FCHAIN, T^2. FCHAIN)	FAND が真の場合の分岐先はすでに埋込み済みなので T^2 の場合だけを写しとる. T^2 が偽の場合の分岐先を T^1 の偽の場合の分岐先リストに連結.

F ::=(E)

F. TCHAIN:=E. TCHAIN; F. FCHAIN:=E. FCHAIN	F も E も真または偽の場合の分岐先は，それぞれ同じである.

F^1 ::=NOT F^2

F^1. TCHAIN:= F^2. FCHAIN; F^1. FCHAIN:= F^2. TCHAIN	NOT であるから F^2 が偽であれば F^1 は真. そこで分岐先を入換える.

F::=I

F.TCHAIN:=NEXTQUAD; F.FCHAIN:=−NEXTQUAD; ENTER(B, I.SEM, 0, 0)	つぎの 4-組を指すべく，くさりの端の値を定め，4-組を生成する.

演習問題　13.6.1

1. 図 13.2 で与えた 9 通りの式を，構文解析し，上述の意味付けルーチンを用いて，ポーランド記号列を生成せよ.

2. 図 13.2 で与えた 9 通りの式に対するポーランド記法を定め，それを生成するように，上の意味付けルーチンを変更せよ.

13.6.2　下向き解析法

比較のために，いまと同じことを再帰的下向き法を用いて行う方法を示そう．下向きに解析を行う構文機構を書こうとしているのであるから，かなりの融通性がある．以下に示す手続き群は，12.4節に述べた方法でプログラムされたもので，議論の必要はほとんどないだろう．これらの手続きは，下のような繰返しを用いた文法（13.6.4）を使ってプログラムしたものである．実際繰返しの代りに右再帰性を用いた文法（13.6.2）を用いる方がずっと容易である（演習問題 4）.

2 個の意味情報 FCHAIN と TCHAIN は，ここでは各ルーチンに対して引数として渡されている.

```
(13.6.4)  Z::=E
          E::=T {OR T}
          T::=F {AND F}
          F::=I | NOT F | (E)
```

PROCEDURE Z; BEGIN INTEGER FCHAIN, TCHAIN; ENTER(:=, 1,, X); E(FCHAIN, TCHAIN); FILLIN(FCHAIN); ENTER(:=, 0,, X); FILLIN(TCHAIN) END	Z ::=E に対する手続き. 最初の 4-組の生成. E の構文解析. 戻ってきたときに偽の場合の分岐先を埋込み X に 0(偽)を代入する 4-組を生成. 真の場合の分岐先を埋込む.

PROCEDURE E(TCHAIN, FCHAIN); INTEGER TCHAIN, FCHAIN; BEGIN INTEGER TC; TCHAIN:=0; FCHAIN:=0; LOOP: T(TC, FCHAIN); MERGE(TCHAIN, TC); IF NXTSYMB = "OR" THEN BEGIN FILLIN(FCHAIN); FCHAIN:=0; SCAN; GO TO LOOP END END	E ::=T {OR T} に対する手続き. 鎖の端を 0 に初期設定. 項(T)の構文解析. 真の場合の分岐先を TCHAIN に連結. まださらに OR があれば，つぎの記号への分岐を偽の場合の分岐先に埋込む. つぎの終端記号をとってきてこの処理を繰返す.

PROCEDURE T(TCHAIN, FCHAIN); INTEGER TCHAIN, FCHAIN; BEGIN INTEGER FC; TCHAIN:=0; FCHAIN:=0; LOOP: F(TCHAIN, FC); MERGE(FCHAIN, FC); IF NXTSYMB = "AND" THEN BEGIN FILLIN(TCHAIN); TCHAIN:=0; SCAN; GO TO LOOP; END END	T ::=F{AND F} に対する手続き. 真，偽への分岐先の鎖の端を初期設定. 因子(F)の構文解析. 偽への分岐先のリストを連結. まださらに AND があるか. あり．真の場合の分岐先を埋込む. つぎの因子(F)を調べに進む.

PROCEDURE F(TCHAIN, FCHAIN); INTEGER TCHAIN, FCHAIN; BEGIN INTEGER K; L: IF NXTSYMB = "(" THEN BEGIN SCAN; E(TCHAIN, FCHAIN); IF NXTSYMB = ")" THEN SCAN ELSE ERROR; GO TO ENDF END;	F ::=I | (E) | NOT F に対する手続き. 最初の記号が開きかっこであれば，式(E)の構文解析を行い，最後が閉じかっこであることを確かめる.
IF NXTSYMB = "NOT" THEN BEGIN SCAN; F(TCHAIN, FCHAIN); K:=TCHAIN; TCHAIN:=FCHAIN; FCHAIN:=K; GO TO ENDF; END;	もしNOTであれば，それにつづく因子(F)を構文解析し，真・偽への分岐先の鎖を入れ換える（真だったものが偽となり，偽だったものが真となる）.
IF NXTSYMB = "I" THEN BEGIN TCHAIN:= NEXTQUAD; FCHAIN:=-NEXTQUAD; ENTER(B, NXTSEM, 0, 0); END ELSE ERROR; ENDF: END	名前の場合. 真の場合の分岐先. 偽の場合の分岐先. 4-組の生成.

演習問題 13.6.2

1. 図13.2で与えた9通りの式を下向き用ルーチンを用いて構文解析せよ.

2. ポーランド記号による論理式の表し方を定め，上述の手続き群を変更してポーランド記法を生成するようにせよ.

3. もしANDの直前にある名前にNOTがついていなければ，4-組（BZ, i, A, 0）を生成して，Aが偽ならば4-組iに分岐するようにすることができる．したがって，この4-組を実行して，Aが真ならば，これにつづく4-組に進む．ポーランド記法や機械語では，これを利用して，より能率のよいプログラムを作ることができる．同様の処理は，ORの直前の名前についても行うことができる．このようなより能率のよいコードを生成するように，上述の手続き群を変更せよ.

4. 上述の再帰的手続き群を，文法（13.6.2）に従うように書直せ.

第14章

実行時における変数の記憶場所の割付

実行時の記憶割付法や実行時のデータ領域の様式が一たん設計されると（第8章参照），コンパイル時における記憶割付作業はかなり簡単なものになる．14.1節では，一般的な構成を概説する．ここではCOMMONやEQUIVALENCEは無視しておく．これが問題になるところでは，注意を与えよう．14.2節では，中間結果用変数への記憶割付を議論する．最後に14.3節でCOMMON文やEQUIVALENCE文にある変数に記憶場所を割付ける問題を扱う．

計算機のうちには中間結果を保持するために便利なレジスタを複数個持っているものがある．これらのレジスタを能率よく利用することも，記憶割付の問題の範疇と考えることもできるが，これは17.5節の目的コードの最適化のところで取扱う方が良いだろう．

14.1 変数に対する番地の割付

まず，コンパイラが現在コンパイル中の手続きごとにデータ領域を1個生成すると仮定しよう．このデータ領域の中にこの手続きで用いられるかげのパラメタも，実引数も，プログラマが自分で書いた変数も（あるいはその変数のテンプレートも）作業用変数も全て入ってしまう．

FORTRANでは，一時に1個のプログラム単位しかコンパイルしない．したがって，ただ1個のデータ領域を維持するだけでよい．ALGOLやPL/Iは他方の極で，コンパイルされる手続きの数には制限がなく，それぞれが一つづつデータ領域を持っている．

一般に，コンパイラは，目的プログラムに必要な全てのデータ領域の表Dを持っており，それぞれのデータ領域に対する表Dの要素には，その領域が，静的であるか動的であるかを示す欄とデータ領域に含まれる記憶場所の大きさ

（個数）を示す欄 LENGTH がある．LENGTH の初期値は0である．

記憶割付ルーチンは，一つの手続きに含まれる変数の番地を一時に割付けてしまう．手続きが一つきまるとこのルーチンの最初の仕事は，データ領域の先頭から DISPLAY の番地（使用している場合）と，かげのパラメタの番地とを割付けることである．データ領域の LENGTH には，これら標準的な，長さの定まった，データが使用する記憶場所の個数を設定する．

つぎは，この手続きの仮引数と，この手続きで宣言されている変数に対する記号表のエントリを処理して，番地を割付ける仕事である．記号表の各エントリに対して，つぎのことを行う．

1. エントリにデータ領域のオフセット LENGTH を割付ける．これは，このデータ領域で固定されていない番地の最初のものである．
2. 記号表のエントリに記述されている仮引数または変数に必要な記憶場所の個数を LENGTH に加える．

この処理は，簡単明瞭であり，これ以上説明する必要はない．要請されるのは，ある変数に必要な記憶場所の数が記号表のエントリから決定できるようになっていることである．言語の中には原始プログラムを全て読んでしまわなければこの情報が，わからないものもある（例えば FORTRAN）．したがって，通常の解析が済んだ後に，実行時の番地を割付けるためのパスが必要になる．

FORTRAN IV WATFOR 360 コンパイラ（Cress ほか参照）は，つぎのような機構を用いている．第1パスは，原始プログラムから機械語を直接生成する主要なパスである．この機械語では，変数を参照する時はその変数を記述している記号表のエントリへのポインタで示している．第2パスは，小さなもので実行時の変数に番地が割付けられ，記号表のエントリに置かれる．つぎに，機械語を走査して，記号表のエントリへのポインタを，対応する実行時の番地に置換える．

使用する前に変数の宣言が必要な言語では，記憶場所の割付は宣言に対する意味付けルーチンで行うことができる．したがって，真の意味で1-パスコンパイラが実現できる．

初期値付きの変数

これまで，ある手続きに対する実行時のデータ領域は，コンパイル時には実際には不要であると仮定してきた．普通は，これでよい．LENGTH だけをみ

て，データ領域の大きさを知ればよいのである．言語の中には，変数に初期値を許すものがある．初期値はコンパイル時に定まる．これを可能にするには，実行時におけるデータ領域の写しを初期値付きで実際に作っておき，それを目的コードの一部に含ませておくのが最も容易な方法である．別の方法（あまり好ましいものではない）は，実行の開始時に初期値を割付けるような代入文の系列を生成しておくことである．

境界を揃える問題

IBM 360 では，あるデータは，2語，1語，半語の境界に揃えてやる必要がある．変数 B1, F1, H1, D1 をこの順に（必要ならば境界に揃えて）並べるとしよう．ここでこれらの変数はそれぞれ 1, 4, 2, 8 バイトを占めるとしこれらの変数の番地はそれぞれ，1, 4, 2, 8 の整数倍でなければならないとしよう．すると記憶場所はつぎのように割付けられる．

バイト	0	B1	バイト	8-9	H1
バイト	1-3	空き	バイト	10-15	空き
バイト	4-7	F1	バイト	16-23	D1

これでは，使われない場所がたくさんできてしまうことが問題である．この問題に対する簡単な解決法は，2語の境界に揃えなければならない変数を最初に割付け，つぎに1語の境界に揃えなければならない変数，つぎは半語という順に割付けるものである．

ブロック構造と記憶場所の節約

ブロック構造を持った言語の実行時管理ルーチンは，各手続きごとに一つづつデータ領域を割付けるものとしよう．そのデータ領域内に，その手続きの全てのブロックの全ての変数に対する記憶場所が割付けられる．したがって，図14.1のプログラムに対しては7個の場所が必要となろう．

しかしながら，ブロック構造を利用すれば記憶場所を節約できる．例えば，図14.1 の A3 と A4 には同じ場所を割付けることができる．A2 と A5 も同様であり，結局5個の記憶場所で足りる．

```
BEGIN REAL A1, B1;
   BEGIN REAL A2,B2;
          BEGIN REAL A3; …END;
          BEGIN REAL A4; … END;
   END;
   BEGIN REAL A5; … END;
END
```

記憶場所	データ
0	A1
1	B1
2	A2, A5
3	B2
4	A3, A4

図 14.1　ブロック構造と記憶割付

この機構は，実行時のデータ領域をスタックのように見立てれば，実現することができる．実行時，ブロックに入ると，そのブロックに属する変数に必要な大きさの記憶場所をスタックにのせる．ブロックから出る時は，同じ記憶場所をスタックから取除く．

コンパイル時の番地割付アルゴリズムは個々のブロックに対して，ブロックの開設順に，記憶場所を割当てる．このアルゴリズムでは，変数を 2 個とスタックを 1 個用いる．変数 CURRENTLENGTH には現在処理中の開設されたブロックでこれまでに割付けられた記憶場所の個数がつねに入っており，変数 LENGTH には，そのデータ領域の記憶場所の最大個数が入っている．両変数の初期値はともに 0 である．スタック S は，現在処理中の個々のブロックに割付けられた記憶場所の先頭番地を貯えるために用いられている．このアルゴリズムは以下のとおり．

1. BLOCK BEGIN : CURRENTLENGTH の値を S にのせる．この値はブロックから出るときとり出される．したがって，並行なブロックは，同じ場所を使用する．つぎに，このブロックで宣言されている各変数(例えば) K につぎのように記憶場所を割付ける．

   K に CURRENTLENGTH の値を番地として割付ける．

   K が必要とする記憶場所の数を CURRENTLENGTH に加える

   最後に，このブロックで必要な作業用変数に同様な方法で記憶場所を割付ける (14.2節参照)
2. BLOCK END : LENGTH : = MAXIMUM(LENGTH, CURRENTLENGTH); スタック S の最上段にある値を取除き CURRENTLENGTH に代入する．

このアルゴリズムの説明のために，図14.1のプログラムをこのアルゴリズムで処理してゆく様子を以下に示す．各ステップの実行前における変数とスタッ

クの内容を示している．

ステップ	現在の長さ	長さ	スタック	変数の番地 A1	B1	A2	B2	A3	A4	A5	つぎのステップ
1	0	0									BLOCK BEGIN 1
2	2	0	0	0	1						BLOCK BEGIN 2
3	4	0	0 2	0	1	2	3				BLOCK BEGIN 3
4	5	0	0 2 4	0	1	2	3	4			BLOCK END 3
5	4	5	0 2	0	1	2	3	4			BLOCK BEGIN 4
6	5	5	0 2 4	0	1	2	3	4	4		BLOCK END 4
7	4	5	0 2	0	1	2	3	4	4		BLOCK END 2
8	2	5	0	0	1	2	3	4	4		BLOCK BEGIN 5
9	3	5	0 2	0	1	2	3	4	4	2	BLOCK END 5
10	2	5	0	0	1	2	3	4	4	2	BLOCK END 1
	0	5		0	1	2	3	4	4	2	

14.2 作業用変数に対する記憶場所の割付

主として式の中間結果を置くために作業用の場所が必要である．ここでは添字付変数の番地や，実引数の値や，関数呼出しの引数の一覧表などを置いておくための一時的な場所も含んでいる．中間結果用変数のための記憶場所は通常その式が現れている手続きあるいはブロックに属するデータ領域の中に割付けられる．各中間結果用変数には，それぞれ別々の場所を割付けることもできるが，これでは記憶場所の無駄遣いとなるので，記憶場所の使用個数を最小にするような記憶割付のアルゴリズムを設計したい．まず，基本ブロック——1列に実行されてしまう命令の系列で途中に分岐もなければ，よそからの飛込みもない系列——だけを考えよう．

つぎの実行文 E:＝A*B＋C*D; F:＝A*B＋1; G:＝C*D に対する 4-組の系列を考えよう：

```
(14.2.1)  (1)  *  A, B, T1          (6)  +  T4, 1, T5
          (2)  *  C, D, T2          (7)  := T5, , F
          (3)  +  T1, T2, T3        (8)  *  C, D, T6
          (4)  := T3, , E           (9)  := T6, , G
          (5)  *  A, B, T4
```

記憶割付だけに限って中間結果用番地 T1, T2, … の割付と参照を ASS, REF とすると，これはつぎのように書くことができる

(14.2.2)	(1)	ASS	T1	(7)	ASS	T4
	(2)	ASS	T2	(8)	REF	T4
	(3)	REF	T2	(9)	ASS	T5
	(4)	REF	T1	(10)	REF	T5
	(5)	ASS	T3	(11)	ASS	T6
	(6)	REF	T3	(12)	REF	T6

中間結果用変数 T_i の範囲 (range) をその変数が最初に定義された点から最後に参照された点の間の演算の系列と定義する．明らかに，共通な範囲を持たない中間結果用変数には同じ記憶場所を割付けることができる．上の例では，T1，T3，T4，T5，T6 には全て同じ場所を割付けることができる．

中間結果用場所の割付アルゴリズムは，機械語のコードが生成された後に働くことが多い．このアルゴリズムは機械語命令を処理するもので，内部表現は処理しない．これは，このアルゴリズムが必ずしも全ての中間結果用変数に記憶場所を割付ける必要がないからである．例えば，(14.2.1) の4-組の系列に対する機械語のコードでは，T5 に対する値は，多分その範囲を通じて累算器か，高速レジスタにのっているだろう．したがって，そのための記憶場所は必要なかろう．これはコードが生成されるまでは不明である．

このように中間結果用変数の記憶割付を，割付 (ASS) と参照 (REF) の系列による個別のパスとして説明した．このアルゴリズム自体は，コード生成部の中に組入れてもコード生成に続く別個のパスとしてもよい．

実行時スタックと作業用変数

実行時の記憶場所 a, $a+1$, $a+2$, $\cdots$ が中間結果用変数に割付けられるとしよう．まず被演算子が単純変数と定数のみからなる式を考えることにしよう．さらに式の評価は，つぎの構文規則に厳密にしたがうものと仮定する：

$$E ::= E + T \mid E - T \mid T$$
$$T ::= T * F \mid T / F \mid F$$
$$F ::= (E) \mid I$$

式 E＋T を評価すると，まず E が評価され，つぎに T，最後に E＋T が評価される．これは，E を評価するのに用いた中間結果用変数は，E の結果ができ上がればもはやいずれも不要となり，T についても同様である．さらに一たん E＋T の計算が終ると，E と T を置いていた中間結果用変数ももはや必要

ない.

ここで注意すべき重要な点は, 2個の中間結果用変数 T_i, T_j の範囲に共通部分がないか, 一方が他方の入れ子になっていることである. 例えば, 式 (A + B) * (C + (D * E)) に対する中間結果用変数の割付 (ASS) と参照 (REF) の系列は

(1) ASS T1 (T1 := A + B)
(2) ASS T2 (T2 := D * E)
(3) REF T2 (C + T2 の評価)
(4) ASS T3 (T3 := C + T2)
(5) REF T3 (T1 * T3 の評価)
(6) REF T1 (T1 * T3 の評価)
(7) ASS T4 (T4 := T1 * T3)

のようになる. (実際, T3 と T1 の参照は, 両者とも本質的には同時に参照されるのだから, REF T1, T3 と書くことができよう.) このように中間結果用変数の範囲が入れ子になっていることから記憶場所 a, $a+1$, $a+2$, … を実行時スタックとして使うことができる. これはポーランド記法になっている式を評価したときと同様である. この操作はコンパイル時につぎのようにして生成される：初期値を0とする変数 L を現在のスタックの大きさを示すために使う. 中間結果用変数に記憶場所を割付け, 演算を順序付ける. 中間結果用変数 T_j に場所を割付ける (ASS) たびに, そこからその変数の範囲が始まり, その変数の実行時の番地に a + L を割付け L を 1 あげる. 中間結果用変数 T_j を最後に参照したところで L を 1 さげる.

上の例で, このアルゴリズムを用いると a を T1 に割付け, $a+1$ を T2, $a+1$ を T3, a を T4 に割付ける. 使用される記憶場所は2個にすぎない.

より一般的な割付法

原始プログラムの内部形式を修正して, 2個の中間結果用変数 T_i と T_j の範囲が重なるようにすると, これまでのアルゴリズムはもはや適用できない. 例えば, コードの最適化のパスでは, (14.2.1) で, 式 A * B と C * D は, 一度だけ計算すればよいことを検知して, (14.2.1) を

(1) * A, B, T1 (5) + T1, 1, T5
(2) * C, D, T2 (6) := T5, , F

(3)　+　T1, T2, T3　　(7)　:=　T2, , G

(4)　:=　T3, , E

と変更することができよう．ここでは，T1 と T2 の範囲は以下に示すように重なり合っている

(14.2.3)	(1)	ASS	T1	(6)	REF	T3
	(2)	ASS	T2	(7)	REF	T1
	(3)	REF	T2	(8)	ASS	T5
	(4)	REF	T1	(9)	REF	T5
	(5)	ASS	T3	(10)	REF	T2

したがって ASS T_i と REF T_i の任意の系列を考えなければならない．Danzig と Reynolds (66) は，記憶場所の使用数を最小にする以下のようなアルゴリズムを証明した：**コンパイル時**スタック S を設ける．これには最初，実行時の番地 $a+n$, $a+n-1$, $\cdots$, $a+1$, a が入っている．"a" が最上段に入る．番地を割付けるために，ASS と REF の操作系列をたどる．ASS T_j で T_j の範囲が新たに始まるので，スタックの最上段にある番地をとってきて T_j に割付ける．最上段の要素は取除く (pop up)．REF T_j の最後の参照（これで T_j の範囲が終る）に対して，もし T_j に記憶場所が割付けられてなければ，T_j は決して割付けられることがないことを意味し，誤りが存在するはずである．もし記憶場所が割付け済みであれば，T_j に割付けられていた番地をスタック S に返してやる．この番地は，もはや必要ない．

このアルゴリズムを (14.2.3) の系列を用いて，以下に例示しよう．第1欄はスタックの内容を示し，第2欄は，第3欄にある操作を行う直前に割付けられている番地を示す．

スタック	割付けられた番地 T1	T2	T3	T5	操　作
$a+3$　$a+2$　$a+1$　a	0	0	0	0	(1)　ASS　T1
$a+3$　$a+2$　$a+1$	a	0	0	0	(2)　ASS　T2
$a+3$　$a+2$	a	$a+1$	0	0	(3)　REF　T2
$a+3$　$a+2$	a	$a+1$	0	0	(4)　REF　T1
$a+3$　$a+2$	a	$a+1$	0	0	(5)　ASS　T3
$a+3$	a	$a+1$	$a+2$	0	(6)　REF　T3
$a+3$　$a+2$	a	$a+1$	$a+2$	0	(7)　REF　T1

$a+3$ $a+2$ a	a $a+1$ $a+2$ 0	(8) ASS T5
$a+3$ $a+2$	a $a+1$ $a+2$ a	(9) REF T5
$a+3$ $a+2$ a	a $a+1$ $a+2$ a	(10) REF T2
$a+3$ $a+2$ a $a+1$	a $a+1$ $a+2$ a	

スタック中の番地は大から小の順に並んでいる必要はないことに注意しよう．実際，作業用として用いることができる記憶場所は，どんな番地の集合であっても構わない．

上に述べたアルゴリズムは，当の REF 操作が同じ作業用の場所に対する最後の参照であることがわかっていることを前提にしている．これによって，各作業用変数のための表の記述にこれ以上の情報を書かなくてよいようにしている．このことは後述しよう．実際には，Danzig と Reynolds (66) のもとのアルゴリズムでは，上の操作系列を**逆順**に処理している．これは，ASS 操作は，各作業用変数に対して，一度しか現れないことを前提としているからである．したがって，逆順に処理すると，最初に REF T_j を処理したときにこれに記憶場所を割付け，唯一の ASS T_j に行き当ったところで，この記憶場所を開放する．

他の作業用変数へのアルゴリズムの適用

基本ブロックの外で使用される作業用変数に対しても記憶場所割付けのアルゴリズムを拡大して容易に適用することができる．例えば，FOR ループ

FOR I:＝A STEP B UNTIL C DO 〈実行文〉

で増分を示す式 B の値を保持するために用いられる作業用変数 T はこのループ用に生成されたコードの全系列が範囲となる．この範囲は，当然のことながら，〈実行文〉を評価するために必要な全ての作業用変数の範囲を含んでいる．

例えば，つぎのような文脈の中に現れる条件式

…A＋(IF B THEN C ELSE D)＋…

に対して，以下のような内部コードを生成できよう．

(1) …
(2) BZ 5, B
(3) :＝C, , T1
(4) BR 6
(5) :＝D, , T1
(6) ＋ A, T1, T2
(7) …

したがって，T1 の範囲は4-組(3)から(6)までとなるので BZ 命令が生成された直後に T1 に番地を割付け，A を T1 に加える時その値を参照するのでその後でこれを解放してやればよい．

11.5節に述べたプログラム・グラフと基本ブロックを用いると，いくつかのブロックにまたがって用いられる作業用変数の範囲を限定するためにつぎのような規則を用いることができる（ここで使用する術語については，18.3 節参照）．いくつかの基本ブロックにまたがって用いられている作業用変数の範囲とは，T がつぎのような性質を持って用いられている各ブロックを含んだ最小の領域 *R* である．その性質とは，*R* の後続ブロックから *R* に戻り，T を参照する(REF T) 任意の道が最初に ASS T の操作を通過することである．実用上は，このような領域 *R* を見付けるのは実際的ではないので，すでに見たように，原始プログラムのレベルでの情報が，範囲を見出すために用いられる．

作業用変数の記述と範囲の指示

各作業用変数の記号表上のエントリ中に変数の範囲に関する情報を貯えておく必要があるかもしれない．3 通りの方法を示そう．

1. 作業用変数のエントリに，この変数が4-組中に現れる回数を計数する欄を設ける（この計数は，4-組を作り上げつつ行い，コードの最適化を行う時まで維持される）．4-組に対するコードを生成する時 4-組の各被演算子あるいは結果が作業用変数の場合その変数の回数欄から 1 を減ずる．もし，この欄の値が 0 となれば，この変数はもはや必要ではなく，この変数に割付けておいた場所は解放できる．
2. 作業用変数のエントリに，この変数が最後に現れる4-組の番号を憶えておく．この4-組に対するコードが生成されると，この作業用変数は不要となる．
3. 4-組を生成（し最適化を）しながら，各作業用変数を参照する4-組を全て 1 次元リストにつないで，このリストの先頭と後尾を，この変数のエントリに記入しておく．例えば，(14.2.3) に対する 4-組が生成された後に

```
(1)  ASS T1 (4)     (6)  REF T3(0)
(2)  ASS T2 (3)     (7)  REF T1(0)
(3)  REF T2(10)     (8)  ASS T5(9)
(4)  REF T1 (7)     (9)  REF T5(0)
(5)  ASS T3 (6)    (10)  REF T2(0)
```

作業用変数	T1	T2	T3	T5
最初の ASS	1	2	5	8
最後の REF	7	10	6	9

が得られているだろう．このためには，各4-組に，リスト上のつぎの要素へのポインタを置くため場所が余分に要ることになる．

このような1次元リストを（作業用変数に限らず）全ての変数に設けるとコードの最適化のために利用することができる（18.4節の小見出し**ループの最適化**以下参照）．さらに，これは，レジスタを能率よく割付けるために利用することもできる（18.4節小見出し**レジスタの割付**以下参照）．3番目に，ここでは，これを作業用変数に記憶場所を割付けるために利用する：すなわちコードを生成しながら，各作業用変数のエントリの記述に，その作業用変数をつぎに参照する4-組の番号を書込む．この4-組のコードはまだ生成されていない．この欄が0であれば，この作業用変数はもはや必要ない．

演習問題 14.2

1. 作業用変数に対して（F_i, L_i）が与えられているとする．ただし，F_i は，変数 T_i に対する最初の割付（ASS）の番号であり，L_i は最後の参照（REF）の番号である．この T_i に記憶場所を割付けるアルゴリズムを設計せよ．使用される記憶場所の数が最小になるようにせよ．ここで，もし $i<j$ ならば $F_i<F_j$ としてよい．すなわち，作業用変数の名前は，出現した順に付けることにする．

14.3 COMMON 変数と EQUIVALENCE 変数

COMMON 変数や EQUIVALENCE で結合された変数に記憶場所を割付ける議論をする前に，これらの変数を含んだ FORTRAN の実行文を概観してみよう．ここで関心があるのは，このような文の構文でもなければ，解析法でもなく，このような文の意味である．構文上可能な全ての変形を述べるつもりはない．

FORTRAN では，サブルーチンは，再帰的ではなく，配列の添字の範囲は全て，コンパイル時に決まってしまうことを，再確認しておこう．したがって，あるサブルーチンあるいは主プログラムに結び付いたデータ領域は STATIC（静的）であり，プログラマが書いた変数を（配列とともに）全て含んでいる．配列は，列方向に貯えられ，行方向ではない．配列の添字の範囲は全てコンパ

イル時に既知となるので，ドープ・ベクトルは不要である．

COMMON ブロック（共通ブロック）

もし FORTRAN のプログラムで A が

```
DIMENSION A (100)
```

と宣言され，C が単純変数であるとしよう．すると

```
COMMON ／A1／A, C, D (200)
```

によって，変数 A (1), A (2), …, A (100), C, D (1), D (2), …, D (200) が A1 と名付けられた，独立のデータ領域（ブロック名 A1 の共通ブロック）の引続いた記憶場所にこの順で現れることを示している．文字列 ″／A1／″ は，省略するか ″//″ で置換えてもよい．この場合，このデータ領域を“無名”共通ブロックと呼ぶ．COMMON 文中に現れる D(200) は，D が要素を 200 個持った配列であることを明記している．こう書いておけば，わざわざ DIMENSION 文で D の寸法を宣言する必要がなくなる．

同じブロック名を有する共通ブロックが同じプログラム中に 2 回以上現れるならば，最初の COMMON 文にある変数のつぎに第二の COMMON 文の変数が続き，つぎに第三，…，となる．したがって，

```
COMMON /A1/ A, B
COMMON /A1/ X, Y
COMMON /A1/ E, D
```

とあれば，これは COMMON /A1/ A, B, X, Y, E, D と書いたのと同等である．同一プログラムのいくつかの COMMON 文の中に同じ名前が 2 度以上現れることはできない．同じ名前が同時に別々の記憶場所に対応することを意味することになるからである．

EQUIVALENCE 文で結合された変数

もし，B と C が DIMENSION 文で

```
DIMENSION B (10, 10), C (10, 10)
```

と宣言されていると，EQUIVALENCE 文

```
EQUIVALENCE(A,B(1,1),C(11))
```

によって，変数 A と B(1,1) と C(1,2) が，同一の記憶場所を共有することを明記している．C(11) は，ここでは，配列 C の11番目の場所を示している．FORTRAN では，配列は列方向に記憶するので——すなわち C(1,1)，C(2,1)，…，C(10,1)，C(1,2)，C(2,2)，…，C(10,2)，…——，11 番目が C(1,2) となる．

別々の配列に属する2個の要素を EQUIVALENCE 文で結合すると，この要素以外の配列要素もそれに従って，それぞれ同一の記憶場所を共有する．配列 B と C の要素がそれぞれ1単位の記憶場所を用いるものならば，上例では B(2,1) と C(2,2) は同じ記憶場所に対応する．同様に，B(3,1) と C(3,2)，B(4,1) と C(4,2) などが対応する．

不都合が生じないかぎり，EQUIVALENCE 文で結合する変数は何個でもよい．

```
EQUIVALENCE(A,B(1,1)),(A,C(1,2)),(B,C)
```

を考えてみよう．A は B(1,1) と C(1,2) と対応付けられているので，これらの3個の変数は同じ記憶場所を共有している．一方，最後の (B,C) によって B(1,1) と C(1,1) が対応付けられることになり不都合が生ずる．何となれば，C(1,1) が C(1,2) と結合することになり，これは不可能だからである．

COMMON 領域にある変数との対応付けも可能である．これによって，COMMON 文に現れなかった変数も共通領域に含まれることになる（この変数と EQUIVALENCE 文で結合されている変数も同様である）．このような EQUIVALENCE によって，COMMON 文に現れた変数の順序が変わるようなことは不可能であるが，COMMON 領域の大きさが拡大することは構わない．

```
COMMON X,Y
DIMENSION C(5)
EQUIVALENCE(X,C)
```

を考えよう．各変数はそれぞれ1記憶単位を占めるものとすると，COMMON 領域の大きさは 5 となる．何となれば C(1) から C(5) までがこの領域に含まれるからである．Y と C(2) が同じ記憶場所を共有することに注意．

COMMON ブロックは，1方向だけしか拡張できない．上例の EQUIVALENCE 文を EQUIVALENCE(X, C(2)) と置換えると誤りとなる．何故ならば，X が COMMON 領域の最初の要素でなければならないからである．

コンパイル時における COMMON ブロック表と COMMON の鎖

コンパイル時において，現在コンパイル中のプログラムで用いられる COMMON ブロック用に表 COMB を使用する．この表の各要素はつぎのような形式である：

NAME	DA	FP	LP	LENGTH

ここで NAME はブロック名（無名 COMMON の場合は空白である），DA はこのブロックに割当てられたデータ領域番号，FP と LP はそれぞれ，この共通ブロックの最初と最後の変数の記号表のエントリへのポインタである．LENGTH は，このブロックに含まれる記憶場所の総計である．

特定の COMMON ブロック中に含まれる変数や配列をたどる簡単な方法は，COMMON 文の中に書かれた順に，それらの記号表のエントリを鎖につないでおくものである．このために各エントリには，ポインタ用に余分な欄 COMP(COMMON Pointer) を設ける．EQUIVALENCE で結合されたデータも同様にして鎖でつなぐ．この場合もポインタ用の欄 EQUIVP を使用する．したがって，記号表のエントリの形式は図 14.2 に示すようになる．この

NAME	COMP	EQUIVP	DA	OFFSET

NAME	―	変数名または配列名
COMP	―	COMMONブロックに属さなければ0，さもなければCOMMONブロック中のつぎのエントリへのポインタ（これが最後のエントリであれば，最初のエントリを指す）
EQUIVP	―	EQUIVALENCEで結合されていなければ0，さもなければ，EQUIVALENCEで結合されているつぎのエントリへのポインタ（環状リストになっている）
DA	―	データ領域番号（割付けが済んでなければ0）
OFFSET	―	データ領域中のオフセットまたはEQUIVALENCEオフセット（後述）

図 14.2　記号表エントリの関連部分

NAME		FP	LP	
A1	0	1	6	0
A2	0	3	4	0

COMB 表

	NAME	COMP
1	A	2
2	B	5
3	C	4
4	D	3
5	E	6
6	F	1

記号表

図 14.3 COMMON ブロックと COMMON の鎖の処理

図では，本節に関係のない属性——型，種類など——の欄は省略した．

両方の鎖とも環状になっていることに注意しよう．すなわち，最後のエントリが最初のエントリを指している．したがって，このポインタは，当の変数が COMMON ブロックに属しているか否か（EQUIVALENCE で結合されているか否か）を示すために使うことができる．ゼロが入っていれば，否であり，ゼロでなければ，属して（結合されて）いることを意味している．図 14.3 は，下記の COMMON 文を処理した後の COMMON ブロック一覧表と COMMON の鎖の様子を示している（ただし，記憶場所の割付は未だ行われていない）．

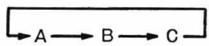
```
COMMON /A1/ A,B
COMMON /A2/ C,D
COMMON /A1/ E,F
```

EQUIVALENCE の鎖

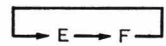
EQUIVALENCE で結合された変数も，環状の鎖（1次元リスト）につないでおき，これには EQUIVP ポインタ欄を用いる．このリスト中の変数の順序は，COMMON 変数の場合とは違って，重要ではない．何故なら，とにかく全ての変数が同一の記憶場所を共有するからである．この事実を使えば，この鎖を最も都合のよいように作ることができる．

EQUIVALENCE で結合された変数は**全て**，同一の鎖の上に現れる．このことは，時々，いくつかの鎖の併合を行わなければならないことを意味してい

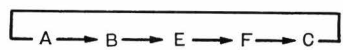
る．例えば，EQUIVALENCE(A, B, C), (E, F, B)を考えてみよう．Bが2度目に現れる直前には，2本の鎖



と



ができている．2番目のBに出会うと，Bはすでに EQUIVALENCE で結合されていることがわかるので，上の2本の鎖を併合して



が得られる．記号表のエントリのOFFSET欄は，EQUIVALENCE で結合された変数（配列）の先頭の要素の記憶場所からの相対位置を示すために用いられる．もし EQUIVALENCE で結合された変数が全て単純変数ならば，これは不要である．しかし

```
EQUIVALENCE(A,B(1),C(2))
```

を考えてみよう．BとCの配列要素はともに1記憶単位を占めることにする．AとBに対する記号表のエントリの OFFSET 欄は0となっている．しかしながら，Cに対するエントリの OFFSET 欄の値は −1で，これは，配列Cの先頭は，変数Aや配列Bより一つだけ前の番地から始まらなければならないからである．さらに，もう一つ

```
EQUIVALENCE(B(2),D)
```

が加ると，B(1) の OFFSET が 0 だから B(2) は1，したがって，D の OFFSET は1となる．

実現法の概要

FORTRAN で COMMON ブロックに属する変数や，EQUIVALENCE で結合された変数に1-パスで番地を実際に割付けることを難かしくしている問題が二つある．第一に変数 A に対する 型宣言文（例えば COMPLEX A）は，A を含んだ COMMON 文（あるいは EQUIVALENCE 文）の後にあってもよい．このことは，COMMON 文あるいは EQUIVALENCE 文の処理時点でその変数の占める記憶場所の大きさが必ずしも既知でないことを意味してい

る．第2点は，配列名に対する DIMENSION 文も EQUIVALENCE 文の中で，その配列要素を書いた後に来てよいことである．例えばつぎの宣言文を考えてみよう．

```
EQUIVALENCE(A, B(1, 2))
DIMENSION B(10, 10)
```

まず EQUIVALENCE 文を処理する時点では，配列 B の寸法が不明なので，配列要素 B(1, 2) の位置を直ちには計算できない．

このために，コンパイラのこれらの処理の部分を作成するには，つぎのような4段のステップを踏まざるを得ない．作成法は，もっと能率よくすることもできようが，ここでは，各ステップの説明を容易にするために，多少能率の悪い方法を提示する．まず概要を述べその後に，4段のステップを多少詳しく述べよう．

1. 原始プログラムを最初に通すとき（第1パス）は，COMMON 変数を全て鎖につなぐ（すなわち，これらの変数用の記号表エントリをつなぐ）EQUIVALENCE の鎖は作らない．それは，B(1, 2) のように，配列の寸法が不明な変数のオフセットを記述するような面倒をさけるためにほかならない．その代りに，事後に処理するために，EQUIVALENCE 文を適当な内部形式にして，別のテーブルにとりわけておく．原始プログラムを全て処理した後には，記号表には必要な情報——型，種類，配列の寸法など——が全て書込まれているものとする．つぎはステップ 2. に進む．
2. EQUIVALENCE の鎖を作成する．同時に OFFSET 欄を埋める．
3. COMMON 変数に実行時の番地を割付け，COMMON 変数と EQUIVALENCE で結合されている変数にも全て番地を割付ける．
4. EQUIVALENCE で結合された変数を含めて，残りの全ての変数に番地を割付ける．

ステップ 1. COMMON 変数を鎖につなぐ

COMMON 変数を鎖につなぐことは，COMMON 文を最初に処理する時点で行う比較的簡単な仕事である．つぎのような COMMON 文を考えよう：

```
COMMON /A1/ A, B(100), C
```

この文の処理は，つぎのような処理に細分される：

1. COMMON ブロック一覧表の中にブロック名 A1 に対するエントリがなければ，つぎのような内容のエントリを新しく設ける．すなわち，NAME = A1, DA = 使用可能な新データ領域番号，FP = LP = LENGTH = 0. P を A1 に対する COMMON ブロック一覧表のエントリへのポインタとする．
2. COMMON 文中の要素を順番に処理する．
   (a) この要素に対して通常の記号表処理を行う（記号表を引いて，もし表中になければ，そのエントリを新しく設ける．型，寸法，などを検査する．もしこの要素が〈名前〉(〈整数〉, 〈整数〉, …, 〈整数〉)，という形式であれば，配列宣言の処理を行う）．ポインタ PS に，記号表のエントリの番地を代入する．
   (b) もし PS.COMP ≠ 0 であれば，この変数はすでに COMMON ブロックに属している．エラーメッセージを出力して，COMMON 文中のつぎの要素の処理に移る．
   (c) PS で指示される記号表のエントリをつぎのようにして鎖につなぐ：
   IF P.FP = 0 THEN P.FP: = PS ELSE P.LP.COMP: = PS;
   PS.COMP: = P.FP;
   (d) COMMON ブロック一覧表の要素を調整する：P.LP: = PS;

ステップ 2. EQUIVALENCE で結合された変数を鎖につなぐ

これは，原始プログラムが一通り処理された後で行い，型や寸法の情報はすでに全て記号表にのっていると仮定している．この処理への入力としては，適当な内部形式となった EQUIVALENCE 文が用いられる．例えば，名前などは，記号表のその名前のエントリへのポインタで表されている．

この処理は COMMON の場合よりも難かしい．それには二つ理由がある：一つは，鎖を併合しなければならないこと，もう一つはオフセットを計算しなければならないことである．しかしながら，EQUIVALENCE の場合はブロック一覧表 COMB のような表は必要ない．その代り，現在作成中の環状リストの適当な要素を指すポインタ変数 P と現在の基底オフセットを値とする整数型変数 BOFFSET だけが必要である．このオフセットを基にして，他の全ての要素のオフセットを計算するものである．BOFFSET は初期値 0 で，鎖を併合したときだけ値が変る．つぎのような文

```
EQUIVALENCE(C, D, A(5), B(1, 3))
```

を処理するために

```
P:= 0; BOFFSET:= 0;
```

を実行し，EQUIVALENCE の鎖中の各要素に対して以下のような処理をする：

1. PS を EQUIVALENCE で結合された要素に対する記号表のエントリへのポインタとする．L をその要素が必要な長さ（実行時の記憶場所の個数）とする．もしこの要素が配列であれば，1配列要素に必要な長さとする．この要素に対するオフセット Z は以下のように計算する：
   (a) もしこの要素が〈名前〉という形式であれば，Z:= BOFFSET.
   (b) もしこの要素が〈名前〉(i) という形式であれば，〈名前〉は配列であり，i が添字の寸法の範囲内にあることを確かめる.† Z:= BOFFSET + L* (1 − i).
   (c) もしこの要素が〈名前〉(i_1, i_2, ⋯, i_n) という形式であれば，〈名前〉が n 次元の配列であり，各添字の値がそれぞれの添字の寸法の範囲内におさまっていることを検査する．配列の中で〈名前〉(i_1, i_2, ⋯, i_n) が j 番目になることを計算する（8.4節参照）．Z:= BOFFSET + L* (1 − j)
2. もし PS がすでに EQUIVALENCE で結合されていれば (PS. EQUIVP ≠ 0) ステップ4. に進む．さもなければステップ3. へ.
3. 新しい要素 PS をつぎのようにして鎖につなぐ：

   IF P = 0 THEN P:= PS ELSE PS. EQUIVP:= P. EQUIVP;
   P. EQUIVP:= PS; PS. OFFSET:= Z. ステップ4. をスキップ.
4. 同一の変数や配列が再度 EQUIVALENCE 文の中に現れると，2本の環状リストを併合して1本にしなければならない.
   (a) 作り直される鎖の中の要素のオフセットを再計算して，旧リスト中の要素のオフセットに揃える．この計算をしながら，自分自身につながる鎖を検査する

---

† 訳注 EQUIVLAENCE 文の中では，多次元の配列も〈名前〉(i) の形で識別できる．したがって‘添字の寸法の範囲’とは‘添字の最大値’を意味する．例えば A (n, m) と宣言されていれば，これは $n×m$ となる．

```
DIFF:= Z − PS.OFFSET;  BOFFSET:= BOFFSET − DIFF;
Q:= P; IF P = 0 THEN BEGIN “新しい鎖はまだ空”:
                        P:= PS; GOTO ステップ 4.(c)
                        END;
LOOP: IF Q = PS THEN BEGIN IF DIFF ≠ 0
                             THEN エラーメッセージの出力;
                        GO TO ステップ 4.(c)
                        END;
        Q.OFFSET:= Q.OFFSET − DIFF; Q1:= Q;
        Q:= Q.EQUIVP; IF Q ≠ P THEN GO TO LOOP;
```

(b)　2本の鎖を併合する．ここで新しい鎖での変数の順番はどうでもよい：

```
Q1.EQUIVP:= PS.EQUIVP; PS.EQUIVP:= P.
```

(c)　この要素の処理は終了．

例として，

```
EQUIVALENCE(A,B(5),C(6)),(D,B(4),E),(F,C(2))
```

を考えよう．図 14.4 は，EQUIVALENCE 文中の各要素がそれぞれ処理された直後に，記号表のエントリがどのように連結されているかを示したものである（ただし，これらの変数は，すでに現れた他の EQUIVALENCE 文中には書かれてなかったものと仮定している）．矢印は EQUIVP 欄を表し，それぞれの変数名のうしろの数字は記号表エントリの中の OFFSET 欄を表している．OFFSET は正にも負にもなりうることに注意．

ステップ	BOFFSET	P	
1	0	A	A, 0
2	0	A	A, 0→B, −4
3	0	A	A, 0→C, −5→B, −4
4	0	D	D, 0　　A, 0→C, −5→B, −4
5	−1	D	D, −1→A, 0→C, −5→B, −4
6	−1	D	D, −1→E, −1→A, 0→C, −5→B, −4
7	0	F	F, 0　　D, −1→E, −1→A, 0→C, −5→B, −4
8	−4	F	F, −4→B, −4→D, −1→E, −1→A, 0→C, −5

図 14.4 EQUIVALENCE の鎖の処理

ステップ 3. COMMON 変数に実行時の番地を割付ける

各 COMMON ブロックに対して，ブロック中の変数の鎖をたどって，番地を割付ける．変数が EQUIVALENCE で結合されていると，この処理を一たん中断して，EQUIVALENCE の鎖の上にある変数の全てに番地を割付ける．このことは，COMMON の鎖の上にある変数は，普通の処理に入る前に番地を割付けられることを意味しよう．この場合，この番地を検査して，これが当然持つべき番地と一致していることを確かめる必要がある．変数 OFFS（初期値は 0）は，つぎの変数に割付けるべきオフセットの値を保持するために用いられている．ポインタ変数 P には，現在処理中の，COMB 表中の COMMON ブロックのエントリの番地が入っている．

番地の割付と同時に COMMON ブロックの記憶域の大きさの決定を行いそれを COMMON ブロック一覧表のそのエントリ中の LENGTH 欄にしまわねばならない．以下に示すアルゴリズムには，“変数 PS の長さ”という語句が現れる．これは，コンパイル時に PS で指示される番地に情報が記述してある変数が実行時に必要とする記憶場所の長さの意味である．もし，これが配列で

あれば，これは配列要素の記憶単位×配列要素の個数を表す．

以下のアルゴリズム中記号表の各欄の名称については図14.2を参照されたい．

PS:=P.FP; IF PS = 0 THEN STOP;	PSはCOMMONの鎖の上にある最
OFFS:=0; P.LENGTH:=0;	初の要素を指す．OFFSは，つぎに
	割付けるべきオフセットの値．
COMMONLOOP: L:=変数PSの長さ;	このループは鎖上の1要素ずつ処理
IF PS.DA = 0 THEN	する．もし0ならば，割付は未だ．
BEGIN IF PS.EQUIVP = 0 THEN	
BEGIN	
PS.DA:=P.DA;	EQUIVALENCEで結合されてい
PS.OFFSET:=OFFS	なければ，この変数に番地を割付け
END	る．
ELSE	EQUIVALENCEであれば，
BEGIN AD:= OFFS−PS.OFFSET;	EQUIVALENCEの鎖の上の変数に
Q:=PS;	番地を割付るために初期設定を行う．
EQUIVLOOP: Q.DA:=P.DA;	変数に番地を割付ける．
Q.OFFSET:=Q.OFFSET+AD;	
IF Q.OFFSET<0 THEN	COMMON領域の拡張する方向を間
エラーメッセージを印刷;	違えた．
L1:=変数Qの長さ;	L1はエントリQにある変数に必要
	な記憶場所の長さ．
P.LENGTH:=MAXIMUM (	COMMON領域の大きさが拡がった
P.LENGTH, Q.OFFSET+L1);	かどうかを見る．
Q:=Q.EQUIVP; IF Q ≠ PS	EQUIVALENCEの鎖上のつぎの
THEN GO TO EQUIVLOOP;	要素を取ってくる．
END END	
ELSE BEGIN IF PS.DA ≠ P.DA OR	割付けが済んでいれば，古い番地と
PS.OFFSET ≠ OFFS	新しい番地が等しいことを確かめる．
THEN エラーメッセージの印刷	
END;	
OFFS:=OFFS+L;	
PS:=PS.COMP; IF PS ≠ P.FP THEN	COMMONの鎖の上のつぎの要素に
GOTO COMMONLOOP;	進む．
P.LENGTH:=MAXIMUM(P.LENGTH, OFFS);	

ステップ4．他の変数に実行時の番地を割付ける

最終のステップは，記号表のエントリを順に見てCOMMONに属さない変数に番地を割付けるものである．もしEQUIVALENCEで結合された変数があれば，その変数がのっているEQUIVALENCEの鎖の上を2度走査しなけ

ればならない．1回目は，鎖上にある全ての要素中の最小のオフセットKを見出すものである．2回目が，番地をそれぞれの要素に与えるものである．鎖上の変数の集合に割付けるべき番地の先頭がOFFSに与えられていれば，各要素に割付ける番地は

OFFS − K ＋（その変数のエントリの OFFSET 欄の値）

で与えられる．したがって，最小のオフセットを持った変数には，OFFS の値が番地として割付けられる．

アルゴリズムは以下のとおり：

1. DA：＝コンパイル中のプログラム単位に対するデータ領域番号;
   OFFS：＝つぎに割付けるべき番地;
   PS：＝記号表の最初のエントリの番地;
2. もし PS. DA ≠ 0 であれば，PS にはすでに番地が割付け済みである．ステップ 4. へ飛ぶ．
3. もし PS. EQUIVP ≠ 0 であれば，PS は EQUIVALENCE で結合されているので，ステップ 3. の(b)のみを実行する．さもなければステップ 3. の(a)のみを実行する．
   (a) 変数 PS の長さLを計算する．
       PS. DA：＝DA; PS. OFFSET：＝OFFS;
       OFFS：＝OFFS ＋ L.
   (b) EQUIVALENCE の鎖上にある全ての変数に番地を割付ける．
       b1. 最小のオフセットKを見付ける：
           Q：＝PS. EQUIVP; K：＝PS. OFFSET;
           WHILE Q ≠ PS DO
           BEGIN K：＝MINIMUM (K, PS. OFFSET);
           Q：＝Q. EQUIVP END;†
       b2. ここで番地を割付ける：MOFFS：＝OFFS − K;
           LOOP：Q. OFFSET：＝MOFFS ＋ Q. OFFSET;
             変数 Q の長さ L を計算する;
             OFFS：＝MAXIMUM (OFFS, Q. OFFSET ＋ L);

† 訳注　MINIMUM は擬 ALGOL の標準関数ではない（補遺 539 ページ参照）が意味は自明であろう．

Q: = Q.EQUIVP; IF Q ≠ PS THEN GO TO LOOP.

4. 処理すべきエントリが記号表中にまだ残っていれば，その番地を PS に代入してステップ 2. へ戻る.

第15章

誤　り　の　回　復

15.1 序　　論

コンパイラは原始プログラム中の誤りはできるだけ多く発見するように努めるべきである．プログラムが正しくコンパイルされるまでに3回，4回あるいはもっと多くの回数，計算機にかけるようなわずらわしいことをプログラマにさせないようにしなければならない．それにもかかわらず，これをうまくやってくれるコンパイラは少いし，このことを取扱った論文はさらに少い．

誤りの回復 (error recovery) という言葉は，コンパイル中に誤りを発見した時に，原始プログラムの解析を続行する方法を決定する処理に対して用いることにする．15.2 節では，意味上の誤りを回復する問題を論ずる．ここでの誤りは大抵名前や式を間違って使用することに関わっている．15.3 節では，構文上の誤りの回復という補足的な問題を，主として形式的構文解析アルゴリズムの中における誤りの自動回復という観点から取扱う．特定の構文解析部の中で，個々の誤りに対する誤りの回復手続きを設計するときには，自動回復の技法が，同じ状況の中でどのような反作用があるかに留意しておく必要がある．このような方法は直接適用できなくても，このように形式的かつ自動的な方法を知っておけばこの種の問題に取組む際の洞察力を養い，コンパイラ全体をより系統だてて作成する助けとすることができる．

どんなに良いと云われる自動的な誤り回復の機構でも広範囲に検査されてはおらず，手で書いた良いものと比較されていないことを知っておくべきであろう．われわれは，このような機構がどの程度よいのかわかっていない．一方，このような機構は，現存するコンパイラに組込まれている直観的な回復手続きの多くのものと比肩しうるものであることは間違いない．

いくつかのコンパイラは，全ての誤りを“訂正”して，コードを生成し，実際にそのプログラムを実行することを試みている．最も著名なのは，CORC (Conway と Maxwell (63), Freeman (63))，CUPL や PL/C (Conway ほか (70)) などに対するコンパイラである．どんなプログラムでも，誤りがいかにあっても実行にまでいってしまう．これは，見ようによっては時間の浪費のように思えるかもしれないが，上述のコンパイラでは，直接実行可能な機械命令を生成しているので，編集結合や実行プログラムのロードに時間がかからない．これがプログラマにとって便利であることは明白であり，どんなに構文上の誤りがあっても，プログラマは，実行時の論理的誤りを発見する機会を得ることになるので，デバッグ用の計算の回数が少なくて済むようになる．カードのたった一つの穿孔ミスで，プログラムの実行を取止めたりせず，その誤りを，正しく訂正するような機会が残されている．

Freeman (63) は，誤りの訂正方法を詳細に述べた唯一の文献である．この文献から，誤り訂正に対する一般的な技法に関する結論を引出すのは困難である．この文献は CORC での訂正の問題だけを扱っており，これはかなり単純な言語である．簡単に説明できる唯一の技法は，綴りの誤りの訂正で，これをしばらく述べよう．

15.2 節や 15.3 節で述べる誤り回復に対する一般的な方法は誤りの訂正に対しても十分用い得ると敢えて言っておこう．もちろん，これらの一般的方法は，より精密にしなければならない．誤りの回復というのは，たった1個の誤りに対してエラーメッセージをいくつも生成するようなことを極力抑えて解析を続行することだけを求めることである．たまたま数個のメッセージが生成されたとしてもそれは実際には問題にならない．一方誤り訂正は，実際に正しいプログラムを生成することを求めている．例えば，間違って使用したエントリを記号表から除いて新しくエントリを挿入する場合，そこに全ての属性が正しく埋められていることを確かめる必要がある．誤り回復では，これは必ずしも必要ない．

誤り訂正の手続きは間違った文を削除してその場所に「この文が削除された」と印刷する出力命令を挿入するようにするのが当然だろう．また，誤りが訂正されると，プログラマに対して，それがどのように訂正されたのかを正確に伝え，それによって彼が自分の実行結果を解釈できるようにしてやらねばならない．普通誤り回復に対しては，これをする必要はない．

綴り字の訂正

コンパイラでの綴り字の訂正を論じた最近の論文に Morgan (70) がある．より詳しい議論やこの問題に関する他の論文の参照のためにこの論文にあたるとよい．

ある名前に綴り字の誤りがあるのではないかとコンパイラが考えるのには，いくつかの状況がある．このような場合コンパイラは，この名前を記号表にある名前と比較してどの名前を書き間違えたのかを決断しなければならない．このような状況のいくつかを簡単に見てみよう．

1. 構文解析の最中には，つぎにくる記号が，予約語の一つであることがわかっていることがよくある．予約語以外の名前が現れたら，これはそのような予約語の一つを書き誤まったのではないかと調べてみるべきである．全ての文が必ずキーワードで始まるような言語がこんな場合の一つである．また，論理式を構文解析していて，AND, OR, GE, LE などの演算が現れるはずだというのも一つの例である．

   またある予約語がくるはずのところで“連接”の誤りも調べることができよう．例えば，BEGIN がくることになってるのに，名前 BEGINA が現れたら，これは BEGIN A と書変えなければならない．
2. 意味分析の最中に，名札として宣言された名前が配列名だけが現れるような文脈の中で用いられるとしよう．するとこの名前は綴り字誤りの可能性があり宣言済みの配列名の表を調べるべきである．同様な状況は，名前の型が前後の文脈から決定できるような場合にしばしば生ずる．
3. 綴り字の誤りのために，ある名前がわずか一度か二度だけ現れて，しかもそれに代入が行われていないか，その値が一度も参照されていないかいずれかの場合がよく起る．この検査は容易に行える．すなわち記号表のエントリに代入カウンタと参照カウンタを設けて，構文解析と意味分析が終った後に，記号表のエントリを走査して，カウンタが0であるような名前はいずれも綴り字を訂正する候補になる．

つぎに究明しなければならない問題は，記号表にあるどの名前を綴り違えたかを決めることである．コンパイラの文脈の中で，これに対する最初の仕事は CORC に関する Freeman (63) の学位論文にある．Freeman の方法は，複雑な得点関数を用意して，ある名前が別のものに書誤る“確率”を推定するものである．この得点関数は一致する文字の個数，1～2文字ずらして一致する

文字の個数，あるいはキーパンチの誤りでよく生ずるもの（0とO，1とIなど）を考慮にいれた上で一致する個数などの情報を用いている．

Freeman の方法は，さらに能率のよい（しかし力は劣る）方法で置換えられた．この方法はプログラム中の綴り間違いのおよそ 80 %がつぎの 4 通りの分類に属するという事実にもとずいている：

1. 1文字の誤り
2. 1文字脱落
3. 1文字余分
4. 隣合う 2 字の入換え

したがって，この種の誤りを調べれば大部分の綴り字の誤りは訂正できる．この方法は Freeman の方法よりずっと速い．Morgan (70) はこれに対する詳細な流れ図を与えている．綴り字訂正手続きの概要はつぎのとおり：

1. 記号表にある名前の集合から，綴り違いによってその名前になりうる名前の全てからなる部分集合を選びだす．
2. この部分集合に含まれる名前各々に対して上の 4 通りの変換を用いて当該の（綴り誤りの）名前になりうるかどうかを調べる．

第 1 ステップの目的は調べなければならない名前の数を制限するためのものである．先に述べたように綴り違えた名前が用いられている文脈はこのために使用できる．明確な検査の一つは名前の長さに関するもので，綴り違いをした名前が n 文字でできいれば，これは $n-1$，n あるいは $n+1$ 文字の名前だけを検査すればよい．$n \leq 2$ のときは，綴り違いの検査はやり甲斐がない．

15.2　意味的な誤りからの回復

本節の目的は意味的な誤り——名前や式の間違った使用法に関する原始プログラムの誤り——を回復する簡単な方法を述べることである．主な問題点はつぎのようなものである：

1. 誤りの回復は，間違った名前あるいは式を“正しい”ものと取換えることから成っている．これは，その誤りが発生した文脈から決定される属性を持った新しいエントリを記号表に挿入し，内部形式プログラムのポインタがこの新しいエントリを指すように変更することによって行われる．

2. 回復法が誤っていたり不完全であったりしたため副次的に生ずるエラーメッセージは，できるかぎり抑制すべきである.
3. 意味的な誤りの回復を行う場所は，1ヶ所，できればERRMES（という名の）ルーチンに局限すべきである．このルーチンはエラーメッセージを印刷する（あるいは後で印刷するためにたくわえておく）ものである．これによって，誤り回復手続きとこれ以外のコンパイラの処理とを分離する効果がある.

扱うべき主要な問題は二つある．すなわち (1) 1個の誤りから派生する余分なメッセージを抑制すること，(2) 同じ誤りが原因となって何度も生ずるエラーメッセージを一度だけですますこと.

副次的メッセージの抑制

ある名前が，間違って使われているときは，エラーメッセージを印刷しそのまま先を続けることができる場合が多い．例えば，代入文 A := B の構文解析を行っており，A が実数型，B が論理型であれば，A と B が代入の両辺で型が合わないというメッセージを印刷して先に進むことができる．何となれば非終端記号〈代入文〉にどんな“意味”も結び付ける必要がなく従ってこれ以上余計なメッセージが印刷されないからである.

一方，添字付変数 A[e_1, ···, e_n] のように，何らかの“構造”がこの名前 A に結び付いている場合を考えてみよう．A が配列名でなく，エラーメッセージを印刷して，添字の構文解析を続けるとしよう．解析を終ったところで，添字の個数を，宣言の添字の個数と比較する．多分これは記号表の A のエントリにあると考えて行う．ところが A は配列名ではないので，きっと副次的なエラーメッセージを生成することになる.

名前に関するこのような副次的なエラーメッセージを抑制するために誤りを回復する簡単な方法は，原始プログラム中でこの名前を“正しい”ものと置換えるものである．記号表には，できるだけ正しい属性を持ったエントリを新しく挿入する．名前としては，原始プログラムに現れるはずのないものとし，さらにこの名前が誤りを訂正するために挿入されたものであることが識別できるものであると仮定する．このようなエントリは間違った名前あるいは式を正すために挿入したのだから CORRECTID エントリと呼ぼう.

もちろん誤りの生じた箇所で全ての属性が分かるというものではないので，

なお余分なメッセージがでるかもしれない．例えば，A[e_1, ・・・, e_n] が現れ，A が実際には配列名でないならば，A[が構文解析された時点で，A が添字を何個持つと考えられるかが不明である．

エラーメッセージを印刷するルーチン ERRMES の存在を仮定しよう．副次的なメッセージを抑制する処置を局限するためには，つぎのようにして，この抑制を ERRMES の中で行う：

> ERRMES の引数として，誤りを起した名前の記号表のエントリへのポインタを与える．ERRMES はエントリの名前を調べてそれが CORRECTID エントリ――前に起きた誤りを正すために挿入されたもの――かどうかを見る．もしそうならば，メッセージは印刷しない．

全ての CORRECTID エントリを新しく作るためのルーチンをただ一つにしておくのもよい考えである．ERRMES ルーチンは，いろいろな呼ばれ方をするので，このルーチンにこの仕事もさせることができる．ERRMES ルーチンについては後述しよう．

同じメッセージは一度で済ませる

同じメッセージを繰返すのは同じ名前が何度も誤って用いられるためである．これは宣言文が正しく解析できなかったためかもしれない．あるいはブロック構造を持った言語で，構文の誤りのために，ブロック構造が完結しないような場合にも生ずる．例えば，

```
BEGIN REAL A; ・・・
   BEGIN BOOLEAN A; ・・・
         BEGIM・・・ END;
         A := A AND B; C := A OR B;
END END
```

のプログラムは，3 行目で BEGIN の綴りを間違えているので，同じ行の END が2行目の BEGIN で始まるブロックを閉じている．4行目の A の参照は全て誤りと見なされ，“A は実数型である．論理型と宣言すべきである”というメッセージが出される．

同じメッセージの繰返しは，簡単に抑制できる．第一は，宣言されていない名前が用いられている場合は，その点でわかる属性とともにその名前を記号表

に挿入すればよい．第二は，ある名前のエントリに，その名前がどんな誤り方をしたか記述する欄を持った一覧表を付属させるものである．もしある名前が不正に用いられると，この一覧表を直ちにたどる．もしそれ以前にも同じような使われ方をしていなければ，メッセージを印刷してこの誤りを一覧表に加えておく．

この手続きもエラーメッセージ・ルーチン ERRMES の中で行われるべきもので，詳細は後述しよう．

プログラマによっては，どんなにたくさんのエラーメッセージが出ようと，とにかく，名前が間違って使われた場所を正確に知りたい者もあるかもしれない．これには，名前を誤って使用した場所を示す行番号の一覧表をその名前の不正な使用を示す要素と一しょに置いておけば出力を減らすことができる．解析が終了したところで，エラーメッセージを一度だけ印刷しあとは，エラーの生じた場所の一覧表を印刷すればよい．

ERRMES ルーチン

このルーチンは，意味的な誤りが生ずるとつねに呼出される．この目的は，メッセージを（できるだけ抑制しながら）印刷しその誤りを正すものである．このルーチンの引数はつぎのとおり：

1. NO：印刷すべきメッセージを区別する番号．
2. ID：誤りの原因となった名前（適用できる場合に限る）．
3. P：正しくない名前を記述しているエントリの番地を含んでいる変数（適用できる場合に限る）．ERRMES は，この誤りを訂正する CORRECTID エントリの番地をPにしまう．
4. T：名前あるいは式が持つべき型を示す引数（誤りが発生した文脈に基くもの）．

引数Tは，誤りを訂正する記号表のエントリに書込むべき属性として用いられる．原始プログラム言語やコンパイラの残りの部分によっては，この属性を記述するのにさらに引数が必要な場合もある．手続き ERRMES の概要は以下のとおり：

1. エラーメッセージ番号 NO が“未宣言の名前”を示すものであれば，ステップ 2. へ進む．それ以外はステップ 3. へ．
2. ID に対する記号表のエントリへ型 T を書入れる．このエントリ番地を

Pに代入し，メッセージを印刷して戻る．

3. Pが以前の名前を正すために挿入されたエントリの番地（CORRECTID エントリの一つ）であれば，そのまま戻る（これによって余分なエラーメッセージを抑制している）．
4. エラーメッセージ NO が，エントリ P の誤った型に関するものでなければ，そのメッセージを印刷して戻る．
5. もしPがすでに同じように誤って使われたものであれば，ステップ7.に進む（この検査は，記号表のPのエントリに基く不正使用の一覧表を走査することによって行う）．
6. エラーメッセージを印刷し，もし P≠0 ならば，エントリPに基いたくさりにこの不正使用をつなぐ．
7. 型Tを持った CORRECTID エントリが存在しなければ，そのエントリを一つ新設し記号表に挿入する．Pが型Tを持った CORRECTID エントリを指すように変更して戻る．

15.3　構文誤りの回復

原始プログラムの構文解析の任意の時点において，プログラムは

(15.3.1)　　xTt

という形式をとる．ここで x はすでに処理を終えた部分を示し，T がつぎに走査すべき記号，t は原始プログラムの残りの部分である．

誤りが生じたとしよう．下向き解析法では，このことは x をカバーするように作られた木の一部が T をもカバーするように拡大できないことを意味している．上向き解析法では，解析法の種類に応じて，x の後部と T の間に順位関係が存在しないか，x のどのような後部も把手を形成しないか，このような何らかの状況が存在する可能性を示している．

この点でプログラムを変更して，この誤りを“調整する”方法を決めなければならない．プログラムの変更は以下のような方法（あるいは，それらの組合せ）によって行うのが最も容易である：

1. T を削除した上で，構文解析を再開する．
2. x と T の間に終端記号の系列 q を挿入し（その結果 $xqTt$ となる）．qTt の頭部を用いて構文解析を始める．上の挿入によって，さらにほか

の誤りが生ずるまで qT の全てを処理することができるようになる.

3. x と T の間に記号列 q を挿入 ($xqTt$ となる) するが, T を用いて構文解析を始める (上向き解析法では, q はスタックにのせられよう).
4. x の後部からいくつかの記号を削除する.

方法3と4は, 良くない回復法で使用すべきではない. これはつぎの理由による. x は処理を終ったのだから, これには意味情報が結合されている可能性がある. q を x に加えたり, x の一部を削除したりすることは, それに従って, その意味情報をも変更しなければならないことを意味し, その上変更を行うのは容易でない. 方法3や4でスタックの内容を変更しなければ, どんな意味をも変更する必要はない. 方法1と2がわれわれにとって主要な誤り回復法である.

終端記号の系列を挿入すること (方法2) によって方法3または4と同等な仕事を遂行できることに注意しよう. 例えば,

$x=$ ···THEN 〈ブロック開始〉　　$Tt=$ ELSE···

であって〈ブロック開始〉を削除したいとしよう. これを削除することは, ブロック構造を持った記号表を調整するなどのブックキーピングを行わねばならないことを意味している. この代りに, 記号 END を挿入すれば

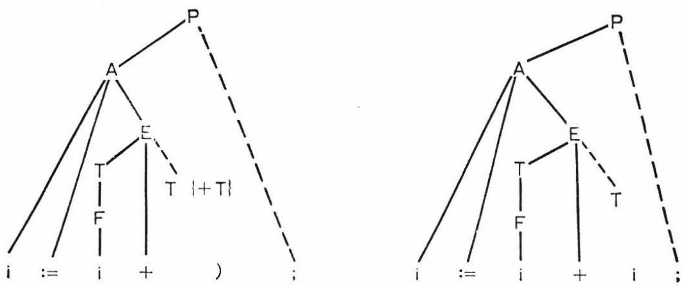
$x=$ ···THEN 〈ブロック開始〉　　$Tt=$ END ELSE···

となる. 通常のコンパイル過程では, 〈ブロック開始〉 END が処理され還元されるにつれて, 全てのブックキーピングを行う.

誤りを考慮した規則を文法に追加するのがよさそうに見えるかもしれない. そこで代入文の左辺の変数が脱落した場合を考えて

〈代入文〉 ::= 　:= E

のような規則を追加するのもよかろう. しかしながら, これによって, 文法はすぐさま大きくかさ張るものになり, ある構文解析アルゴリズムに対して適切な形に変形するのが困難になってくる. 生成言語 (第7章参照) は, この観点からすると, 使うのに持ってこいの言語である. 実際にプログラムをしているのは自分だから, 自分で誤りを考慮して, 随意に生成規則が追加できる.



(a) 誤りの発生点　　　(b) 誤り回復後

図 15.1　下向き解析における誤り回復の例 (1)

下向き解析法における誤りの回復

ここで Irons (63 b) の技法を説明しよう．つぎの文法を用いる：

(15.3.2)　　P ::= A;

A ::= i := E

E ::= T {+T}

T ::= F {* F}

F ::= i | (E)

逆戻りなしで働く下向き法を仮定する．これは可能な構文解析を並列に進めて行き，死んだ端に行き当ったものは落としてゆくか，解析の各段で適用すべき正しい規則を弁別することができるように何らかの文脈を使うかする．

構文解析の任意の段階で，一つあるいはそれ以上の木が，不完全な枝を持ったままで構成されている．例えば，**図 15.1 a** では，実線が部分的に完成した木を示し，点線は P とか E とか名付けられた枝ができるはずだということを示している．

U と名付けられた不完全な枝は

$$U ::= X_1X_2\cdots X_{i-1}X_i\cdots X_n$$

という規則を適用することに対応する．ただし $X_0\cdots X_{i-1}$ はこの枝ですでに完成した部分を示し $X_i\cdots X_n$ が未完の部分を示す．図 15.1 a では，P と名付けられた未完の枝は，規則 P ::= A; を適用することに対応しており，″;″ がこの枝の未完の部分である．E と名付けられた未完の枝は規則 E ::= T {+T} を適用することに対応している．枝を完成させるには，一つの T と

それに任意個の ″＋T″ が続くことが必要である．未完の部分は，したがって T{＋T} である．

つぎの例として規則

$$U ::= (A \mid B\ C \mid D\ E)\ (F \mid G)$$

を考えよう．ここでかっこは超言語記号である．B が枝の完成部分とすると，未完の部分は $C(F|G)$ である．何となれば CF か CG のいずれかで右辺が完結するからである．

このような枝の未完の部分が誤りの回復において大きな役割を演ずる．結局，これによって，原始プログラム中つぎに何が現れ得るか，現れるべきかを知ることができる．

さて，構文解析の最中に誤りが生じたと仮定しよう．これまで部分的にでき上った構文の木は，これ以上作り上げてゆくことはできない．そこでつぎのような誤り回復の手を打つ：

1. 未完の枝の未完の部分にある記号の一覧表 L を作る．
2. Tt の先頭の記号 T を調べては捨てて（新しくできた記号列を改めて Tt とする）L 表中にある適当な U に対して $U \overset{*}{\Rightarrow} T\cdots$ ($U=T$ または $U \overset{+}{\Rightarrow} T\cdots$) となるような T が見付かるまで繰返す．
3. ステップ 2. で記号 U を L 表に置かせた未完の枝を決定する．
4. 終端記号からなる系列 q を T の直前に挿入すると，構文解析を継続して T がステップ 3. の未完の枝に正しく連らなることができるように選ぶ．これはステップ 3. の未完の枝を試してそれによって決まる部分木の全ての未完の枝を調べることによって行われる．このような未完の枝のそれぞれに対して，終端記号の系列が生成されてこれを完成し，このような記号列が連接されて q を形成する．
5. 記号列 q は T の直前に挿入され q の先頭の記号をつぎに来る最初の記号として構文解析を続行する．

例えば，図 15.1 a に示したような構文解析を考えよう．″)″ が入って来るところで誤りが生じた．L 表は {i, T, ＋} である．誤り回復のステップ 2. は ″)″ を捨てることになる．″;″ を L 表に置く未完の枝は P ::= A; である．したがって記号列 q を挿入して枝 E ::= T {＋T} を完成させなければならない．挿入すべき最も簡単な記号列は名前 i である（図 15.1 b 参照）．

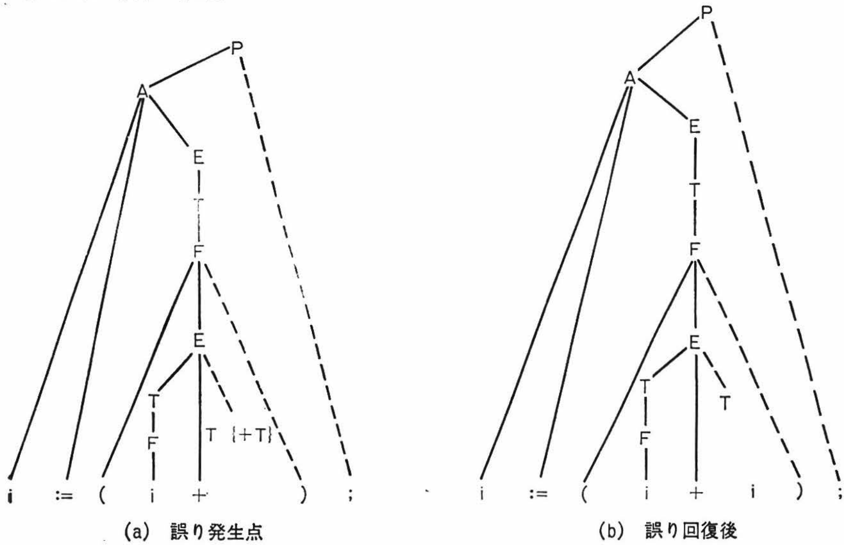


図 15.2　下向き解析における誤り回復の例 (2)

図 15.2 は，回復の仕方を決めるのにこの機構が“大域的な”文脈を用いる有様を説明している．図 15.2 a にある誤りは 図 15.1 a のものと同じように見える：″+″ のつぎに ″)″ がつづいている．しかしながら，L = {;,), +, T} であり，今度はステップ 2. で ″)″ は捨てられない．″)″ を L に置かせている未完の枝は F ::= (E) である．″)″ をこの枝と結び付けるためには，枝 E ::= T{ + T} を完成する記号列を挿入しなければならず，これは再び i である (図 15.2 b 参照)．

開きかっこは誤りの生じた場所からずっと離れているが，それでも誤りの回復の決定に用いられていることに注意しよう．未完の枝の全てが誤り回復でそれぞれの役割を演じている．

再帰的下向き法（再帰的手続きを用いる下向き解析法，4.3 節参照）を用いると，上に述べた下向き誤り回復技法を用いることができるので具合がよい．しかしながら，部分的にでき上った構文の木は，陽に表現されていないので，これが完全に可能であるわけではない．回復法の一つはつぎのようなものである．

ある再帰的なルーチンが誤りを検出すると，エラーメッセージを印刷する．

続いて，つぎに続く原始プログラム記号 T に応じてつぎの二者択一を行う．一つはその誤りを正すものを何か挿入して続行するか，もう一つは呼出したルーチンに，誤りの表示を伴って，戻るかである．例えば，

F ::= (E) | i

に対するルーチンが "(" か "i" がないことを知ったら，"i" を挿入して先を続けることができる．このルーチンが "(E" を見たが閉じかっこが見出せないとすると，それがそこにあると仮定して戻ることができる．

今ここで行っているのは本質的に自動誤り回復法が行っていることと同じものである．実行の各点で，現在実行中の再帰的手続きがそれぞれ，その木の未完の枝を表している．今活きている手続きがつぎに来る原始プログラム記号とそれが表す未完の枝に応じて，できるかぎり最善の回復処置を講じる．それができない場合は，それを呼出したルーチンに不成功を告げる．するとこのルーチンが同じように回復の努力をする．

いずれかの点でこの処理過程は停止すべきである．〈実行文〉とか〈複合文〉に対するような“重要な”ルーチンは，呼出し側に戻るべきでなく，原始プログラム記号を回復可能なところまで——END とかセミコロンとか，あるいは新しい文が始るのがはっきりわかるところまで——飛ばすべきである．

上向き解析法における誤りの回復

下向き誤り回復法のうまい点は，それまでに部分的にでき上っている木の上に，原始プログラム中でそのつぎに現れるべき記号に関する有用な情報がたくさんのっていることである．これらの情報は上向き法では直ちに使用可能というわけには行かない．ここでは xTt が得られているだけである．ただし x はすでに処理済みの文形式の一部である．したがって，全ての未完の枝といった大域的な文脈を簡単に使用することができず，誤りのあった個所近辺の局所的な文脈に頼らざるを得ない．

つぎの方法は，XPL コンパイラ作成システム (McKeeman ほか (70) 参照) で用いられているもので，自動上向き構文解析法の多くで用いられている技法と同じものである．これが素朴な方法であるのは間違いない．

コンパイラの書き手は ";" や "END" のような“重要な”記号を入れた STOPIT という配列を初めに用意する．誤りが検出されると，つぎのような処置をとる：

1. Tt にある記号をつぎつぎと調べて STOPIT にある記号が見付かるまで捨去る.
2. 新しい記号 T を持って来て，x の後部をなす記号を調べては捨去り，この T が x の残った部分に正しくつながるまで続ける.

上のステップは構文と意味を分離しておく（x を変更しない）というわれわれの原則をくずしていることに注意．この方法は $x=x_2x_1$ として，ある規則の右辺の頭部をなす，先頭の数個の記号 x_1 を使用して，行列 M の第 i 行を拾い上げ，走査したばかりの記号 T が列の番号 j を決定する．配列要素 $M(i, j)$ は，サブルーチンの先頭番地を示し，これは T をスタックにのせ，つぎの記号を走査するか，還元を行うかのいずれかを実行するものである．普通この行列の半分以上がありえない組合せ（x_1, T）なので，各種のあやまり条件を検出できる点が具合がよい.

われわれの念頭にある誤り回復機構とは，文法が与えられると，誤り回復サブルーチン群を自動的に構成し配列要素にそれらの名前を埋めてゆく自動作成機（constructor）を使用するものだろう（もしそのような自動作成機が手に入らなければ，コンパイラ作成者は，自分用の特別な誤り回復機構を工夫しなければならない．ただし下記の規則に留意すべきである)．したがって，誤り回復ルーチンは，文脈に基いて回復を図る方法を工夫する必要がなくなり，前以って決めておいた，x_1 と T の組合せだけによって回復を行う.

T を削除するか，x_1 と T の間に終端記号からなる系列 q を挿入するかによってのみ回復を図ることを思い起こすと，この自動作成機は，与えられた（x_1, T）の対に対して回復を図る方法を決定するためにつぎのようなごく自然な方法を使うことができよう.

1. 文法中に $U ::= x_1zT\cdots$ という規則が存在すれば，$z \overset{*}{\Rightarrow} q$ となるような終端記号列 q を挿入すべきである.
2. $U ::= x_1V\cdots$ という規則が存在すれば（ただし $V \overset{+}{\Rightarrow} zT\cdots$），$z \overset{*}{\Rightarrow} q$ となるような終端記号列 q を挿入すべきである.
3. $V \overset{+}{\Rightarrow} \cdots Wz_2$ かつ $W ::= x_1z_1$ が規則で，$U ::= \cdots VT\cdots$ という規則が存在すれば，$z=z_1z_2 \overset{*}{\Rightarrow} q$ となるような終端記号 q を挿入すべきである.
4. 上の規則がいずれも適用できなければ，T は削除すべきである.

上のような規則は，下向き解析法で用いられているものと極めて類似しているが，決定を下すために用いる文脈が少なくなっている．このことは誤り回復が下向きの場合と同様には行えないことを意味している．下記の表で記号列 q の決定を説明しよう．ALGOL 型言語で慣用の構文を仮定している．CS は〈複合文〉を表し，SL は〈文の並び〉を表す．

場合	x_1	T	規則	z	q
1	(	)	F ::= (E)	E	i
2	BEGIN	:=	CS ::= BEGIN SL END	i	i
			SL $\stackrel{*}{\Rightarrow}$ i := E		
3	E *	)	F ::= (E)	F	i
			E $\stackrel{*}{\Rightarrow}$ T		
			T ::= T * F		
4	ARRAY	STEP		(STEP を削除)	

この自動的な方法はまだ使用も検査もしていないので，概要を示すにあたって，以下のような未解決な重要問題がいくつか残されている．すなわち，もし2個以上の規則が適用できる場合は，どれを採用すべきか？ $z \stackrel{*}{\Rightarrow} q$ となるような q のうちどれを挿入すべきか？ ある挿入を行った結果，無限の挿入の系列を引起こすことにならないか？ しかしながら，この技法は，直観的な形で ALCOR ILLINOIS 7090 コンパイラ (Gries ほか (65) 参照) に用いられ成功している．

第16章

インタプリタ

インタプリタ (interpreter) という言葉は，つぎの二つの機能を遂行するプログラムに対して用いる：

1. 原始言語（例えば ALGOL）で書かれた原始プログラムを翻訳して内部形式に変換し，
2. この内部形式で書かれたプログラムを実行する（解釈実行する，模擬実験する）．

インタプリタの第一の部分は，複数パスを持ったコンパイラの最初の部分と同じなので，これを“コンパイラ”と呼ぼう．ここで翻訳されてでき上る内部形式は，第二の部分——真のインタプリタ——ができるかぎり能率よく働くようにに設計しておくべきである．ここでは，ポーランド記法が使われることが多く，これが議論の焦点である．これは，11.2 節で議論したように解釈されるので読者は先を読む前にもう一度，この節を読返してみるとよい．本章は，実際の作成法の詳細に議論を限定している．

インタプリタという言葉は，上述の二つの部分全体の処理を引用してはいるけれども，以下ではこの言葉は，後の部分を引くためだけに用いることにする．何となれば，後の部分こそ内部形式が解釈実行されるところだからである．この第二の部分は“解釈実行時”に現れる．

インタプリタは，それに同等な機械語のプログラムに比べると実行時間がずっと遅い．したがって，計算機時間の大部分を実行に費やすような計算結果を生産する環境では用いるべきではない．計算機時間の大部分を原始プログラムのデバッグに費やすような教育目的の環境にこそ解釈実行が有用である．これから分るように，コンパイラよりインタプリタの方が，実行時のデバッグ用の道具立てを行うのが容易である．

インタプリタは，原始言語にそのまま適用することもできて，これは，実行時に，実行の重要な部分が，標準のシステムルーチン群の中で費やされ，実際に目的プログラムの命令群を実行するのに要する時間の方が少ない場合である．これには，例えば，行列演算用言語がある．ここでは，大部分の演算は，標準の，すでにアセンブル済みのシステムルーチン群を呼出して行列演算を行っている．ほかにも SNOBOL のような文字処理用言語などがある．

機構の概略

計算機の内部では，原始プログラムをポーランド記法にして整数型配列Pに貯える．解釈実行中には，整数 p (初期値 1) によって，現在処理中の記号が入っているPの指標を表す．したがって，p は“実行順序カウンタ”の役をしている．11.2 節に述べたように，ポーランド記法で記されたプログラムを実行するにはスタック S (その指標を i とする) を用いる．スタックの初期状態は空である ($i=0$)．

インタプリタの主要な部分は，CASE 文であり，CASE 文中の各部分実行文はポーランド記法で書かれたプログラムの演算子と被演算子の型の可能な組合せに一つづつ対応する．したがって，各ステップごとに，P (p) が CASE 文の部分実行文の一つを決定し，それを実行する．図 16.1 は，11.2 節で与えた演算子と被演算子の表現法（以下に再掲する）についてこれを説明している：

```
 1 = 定数                     11 = BLOCK（ブロックの開始）
 2 = 名前                     12 = BLCKEND（ブロックの閉鎖）
 6 = SUBS（添字計算）          13 = ADEC（配列宣言）
 7 = :=（代入）               14 = +
 8 = BMZ（負，0 のとき分岐）    15 = *
 9 = BR（単純分岐）             16 = −
10 = BRL（名札へ分岐）
```

ここでは，大部分の部分実行文は，普通の文章で記述する．インタプリタの概略の構造を示すのがここの目的だからである．他の2点も図で説明している．部分実行文8は，その2個の被演算子をスタック S から取除いている．各部分実行文ともこれを行い，さらに演算結果があればそれをスタック S にのせなければならない．第二に，部分実行文が実行されると，実行順序カウンタは

```
NEXT:
CASE P (p)  OF
   1:BEGIN  p := p+1;  "P(p)をスタックSにのせる"  END;
   2:BEGIN  p := p+1;  "P(p)をスタックSにのせる"  END;
   3: ;
   4: ;
   5: ;
   6:BEGIN  "添字の番地計算を行う"  END;
   7:BEGIN  "代入を行う"  END;
   8:BEGIN  i:= i-2;  IF  S(i+1) < 0  THEN
                     BEGIN  p := S(i+2);GOTO  NEXT  END END;
   9:BEGIN  i := i-1;  p := S(i+1);  GO  TO  NEXT  END;
  10:BEGIN  "名札への分岐を行う"  END;
  11:BEGIN  "ブロックの開設処理を行う"  END;
  12:BEGIN  "ブロックの閉鎖処理を行う"  END;
  13:BEGIN  "配列宣言を行う"  END;
  14:BEGIN  "加算を行う"  END;
  15:BEGIN  "乗算を行う"  END;
  16:BEGIN  "減算を行う"  END;
ENDCASE;
p:= p+1; GOTO NEXT;
```

図16.1 インタプリタの概要

1だけ増して，つぎの記号の処理に進めるようにすることに注意しよう．部分実行文1と2では特別にpに1加えているのは，われわれの表現法では被演算子が2語の場所をとっているからである．分岐に対応する部分実行文はp自身を変更し，直接NEXTへ分岐してつぎの記号を処理しなければならない．例えば BMZ に対する部分実行文8がそれである．

スタック S 上の値

しばらくの間，原始言語で扱うデータの型は INTEGER のみと仮定しよう．複数の型と型変換に伴う問題は後述しよう．さらに記号表は，解釈実行時にも使用できると仮定しよう．スタックSにはつぎの3種のデータがのる可能性がある：

1. 整数値を持ったデータ
2. 記号表のエントリへのポインタ
3. 変数の番地

″A B +″ の演算を評価すると第1種のデータ，整数値がスタック上にのる．他の2種は，例えばポーランド記法で

〈式〉…〈式〉 A SUBS

のような添字付変数を評価するときに現れる．ただし A は，配列の記号表上のエントリへのポインタである．スタック上では，これらの〈式〉は整数値で表されている．ポインタ A は，演算子 SUBS が，添字の個数を決定し添字付変数の番地を計算するために必要とされる．計算された番地が，第3種のデータで，SUBS 演算の結果としてスタック上に置かれる．

スタックの要素にはそれぞれ欄が二つあり，KIND，VALUE と称する．S(i).KIND は 1,2,3 をとり，これはそれぞれ，S(i).VALUE が，整数値か，記号表のエントリの番地か，変数の番地かに対応している．各演算子は，実行のときに，スタック上にあるその被演算子を検査して，その演算を行う前に，正しい KIND に変換しなければならない．

例えば，演算はつぎのようなステップを踏むことになろう（被演算子は S(i) と S($i-1$) に入っている）：

1. もし S(i).KIND＝2 であれば，S(i).VALUE にある番地の指す記号表のエントリで記述されている変数の値を S(i).VALUE に代入する．もし S(i).KIND＝3 であれば，S(i).VALUE にある番地の示す変数の値を S(i).VALUE に代入する．
2. ステップ 1. と同じ操作を S($i-1$) に対して施す．
3. S($i-1$). VALUE：＝S($i-1$). VALUE＊S(i). VALUE; S(i). KIND：＝1（乗算を実行し，種類を定める）．
4. $i := i-1$（スタックの調整）．

上述の方法が，解釈実行に多くの時間を要するのは明らかである．どの演算子もいちいち被演算子を検査して，結果の種類を設定しなければならない．もしコンパイラが，演算子の各被演算子の KIND と，その結果の KIND をも決定することができれば，コンパイラがポーランド記法のプログラム中に変換を行う演算子を挿入するようにしたもっと能率のよいシステムを設計することができる．ここでは，演算を実行しようとするときに，その被演算子はすでに正しい種類になっている．

つぎのような三つの変換演算子を定義しよう．

1. CVPV. スタックの最上段の被演算子を変換 (convert) して記号表のエントリへのポインタ (pointer) からそのエントリで記述されている値 (value) にする.
2. CVPA. スタックの最上段の被演算子を変換 (convert) して記号表のエントリへのポインタ (pointer) からそのデータの実行時の番地 (adderess) にする.
3. CVAV. スタックの最上段の被演算子を変換 (convert) してデータの実行時の番地 (address) をその番地に入っている値 (value) にする.

さらに，ポーランド記法になっている名前を三つの範疇に分類する. 名前 I が2語の対

(1, 記号表のエントリへのポインタ)

で表せたことを想起しよう.† この第1語に番号 2, 3, 4 のいずれかを置いて，その名前の**処理方法**を知るために用いる：

2. スタック上に記号表のエントリへのポインタを置く. これを P：I という記号で表す.
3. スタック上に変数の番地を置く. これを A：I という記号で表す.
4. スタック上に記号表のエントリで記述される変数の値を置く. これを V：I と表す.

例えば，代入文 C：＝B＋A(I) (ポーランド記号で C B I A SUBS ＋ ：＝) をより明白な上述の記法で書くとつぎのようになる：

A：C	C の番地は結果を代入するために必要
V：B	B の値を加算のために
V：I	I の値 (添字)
P：A	配列 A の記号表のエントリへのポインタ
SUBS	A(I) の番地を計算してスタック上に
CVAV	A(I) の値を加算のために得る
＋	加算，結果をスタックの上に
：＝	C に結果を代入

† 訳注 図 11.2 参照

一般に，後に述べた方法が好まれている：コンパイル時にできることを，実行時まで持ち越すな！　ここでは，KIND 欄は不要で，それは各演算の期待すべきものが正確に分っているからである．

被演算子の変換

被演算子の型の変換には，二つの似たような考え方がある：スタックの要素ごとに，値の型を示すための欄を設け，各演算子にこの欄を検査させるか，必要な変換を陽に記述するかである．原始言語が整数型と実数型の2通りの型を持ち，I と J が整数型変数で A と B が実数型変数としよう．もし I := A * B + J を

```
I A B * J + :=
```

と表すならば，スタック上の要素の型を示すために TYPE 欄を設ける必要がある（TYPE 欄と KIND 欄は，併合して一つの欄にすることもできよう）．もし原始言語が許せば，

```
I A B *R J CVIR +R CVRI :=I
```

の方が好ましい表現法である．ここで CVIR は単項演算子で，スタックの最上段の要素を整数型から実数型に変換するものであり，CVRI は同様に実数型から整数型への変換である．さらに原始プログラムのいくつかの演算子に対して複数のポーランド記法が必要になることに注意しよう．上の例では，乗算は *R（実数の乗算）と *I（整数の乗算）で表している．

コンパイルを簡単にするために，2個の演算子 CVIR2 と CVRI2 も必要で，これらは，スタックの上から2番目の値を変換するものである．したがって，I := J + A * B は

```
I J A B *R CVIR2 +R CVRI :=I
```

と表わしてよかろう．これは内部形式を生成する道程のゆえに必要である．すなわち，われわれには I J A B *R が生成されるまで J を変換しなければならないことがわからない．

型変換を陽に挿入する方法の方が，内部表現を貯えるための記憶場所をたくさん使用し，陽に示さない方法に比べてほとんど2倍の演算が用いられている（原始言語が2種の型を持っていると仮定して）．さらにコンパイラの側からも

より多くの仕事が要求される．しかしながら，各変数や中間結果の型を，コンパイル時に決定することができれば，実行時間がずっと速くなることから，陽に示す方法の方が好ましい．

もちろん，変数の型が実行時に変化するような言語もいくつかある（例 SNOBOL）．この場合，陽に示さない方法を使用しなければならない．さらに，現在の変数の型を絶えず追いかけるために記号表のエントリが，使用できるようになっていなければならない．この点で，記号表のエントリはその変数に対するテンプレートのように振舞っている．

多くの場合，変数の型は，その変数が参照されている実行文の前後の文脈から決定できるものが大部分である．このような場合はできるだけ陽に示す方法を用いるのが賢明で，必要な場合にのみ陽に示さない方法に切換えるのがよい．

解釈実行時の記憶域の管理

解釈実行時の記憶域の管理は，8章で述べた方法とほとんど差異がない．もっとも，インタプリタを記述している言語に合うように細かい点を二，三変更することは必要である．例えば，インタプリタの記述に ALGOL を用い，原始言語が整数型と実数型を持っていれば，二つの配列を用いて，実行時の変数の値を保持するための実行時スタックにする必要がある．一つは整数用で，もう一つは実数用である．任意の変数の番地は，二つの配列の一方の要素への指標になる．ALGOL にはポインタ型変数がないので，なにごともこれらの二つの配列への指標を通じて行わなければならない．もし，インタプリタを FORTRAN で記述すれば，二つの配列を EQUIVALENCE で結合し，実質的に配列を一つだけにして，いくらかのブックキーピングを節約することができる．

原始言語の手続きが再帰的でなければ，各単純変数の現在の値を，その変数の記号表のエントリにそのまましまうことができる．この場合，実行時の番地は実際には必要ない．

記号表のダンプ

インタプリタがコンパイラに対して持っている利点の一つに，プログラマに対してデバッグ用の道具を提供しやすいということがある．例えば，解釈実行時の手続き PRINTVALUE (P) を用意するのはかなり簡単である．この手

続きは，記号表のエントリの番地 P を与えると，その原始プログラム上の名前と現在の値を並べて印刷するものである．現在の値は，変数の現在の型に応じて整数，10 進 の 浮動小数点数そのほかである．

原始言語が，再帰的手続きを含まず，したがって実行時の記憶域の管理がかなり簡単であれば，PRINTVALUE は記号表のダンプ（内容の印刷）に用いることができる．解釈実行時に誤りが発生したときはいつでも，記号表をたどりながら現在実行中の手続きの変数に対応する記号表のエントリをそれぞれ引数として PRINTVALUE を呼出す．

ここで重要なのは，ダンプが原始プログラムの文字の状態で行われる点である．誤りが生じた時点での記憶の内容をそのまま 2 進や，8 進や 16 進符号でぶちまけるのではない．このように文字によるダンプは，普通のコンパイラにも備えることができる（備えるべきである）が，インタプリタの場合より難かしい（第 21 章参照）．

ブロック構造や再帰的手続きを持った言語では，全ての変数とその現在の値を文字の形でダンプするのは，プログラムが多少大変になる．この機能を達成するには，ブロック構造を記述し，各ブロックの記号表のエントリがある場所を示すブロック一覧表が，利用できるようになっていなければならない．第二に，解釈実行時の任意の時点で，現在のブロックを知ることが可能でなければならない．このためには，各ブロック終了の演算子のところにそのブロックの番号を（ポーランド記法で書かれたプログラムの間に）挿入してやればよい．すると，$P(p)$ の記号が現れるブロックを探すには，$P(p+1)$，$P(p+2)$ … と BLCKEND 演算子が見付かるまで走査を続ければよい．

第三に，ダンプを遂行する手続きは，実行時のスタックの中をのぞいて，ある手続きのデータ領域がどこにあるか決めることができなければならない．手続きのかげのパラメタを探し出すのにこれが必要である．

ダンプが $P(p)$ の演算の時点で生じると仮定して，ダンプの手続きのあらましをここで示そう．詳細は読者に任せる．考え方は，現在実行中の手続きの中で宣言された変数を印刷し，この手続きを呼んでいる手続き中の変数を印刷しというようにして，最後に主プログラムの変数を印刷するというものである．

1. $P(p)$ から始めて，$P(p+1)$，$P(p+2)$，… の順に BLCKEND を探す．見付かれば，ブロックの番号を B に代入する．

2. ブロック B 中の全ての変数を印刷する.
3. ブロック B を囲むブロックの番号を B に代入.
4. もしブロック B が通常のブロックであればステップ 2. に戻る. もし手続きブロックであればステップ 5. へ. もし 0 であれば(これ以上ブロックはないので)終了.
5. 手続き B に仮引数があればその値を印刷する.
6. 実行時のスタックを用いて, この手続きを呼出した場所を探し出す(この手続のかげのパラメタを調べる). あたかも RETURN 演算子が実行されたかのように実行時スタックを書変える. 実行順序カウンタ p を戻り先番地に設定し, ステップ 1. に戻る.

この機構を用いているコンパイラがいくつかある. これに伴ってコンパイラで生ずる問題は第 21 章で簡単に論じる (Bayer ほか (67) も参照せよ). SPL インタプリタもこの機構を用いている (McKeeman ほか (70) 参照).

その他のデバッグ用の道具立て

解釈実行時のエラーメッセージはいずれも誤りが生じた場所を原始プログラムの行で示すべきであろう. これは, 行番号演算子をポーランド記法のプログラムの間に挿入することによって簡単に達成できる(この演算子は NO-OPS すなわち実行はなにもしない). 各プログラムは, 行番号演算子のあとに続く. これらの番号は, ブロック閉鎖演算子を記号表のダンプで用いたと同様な方法で使用することができる.

この目的のためならば, IBM の PL/I コンパイラ (水準 F) でしているように文の番号を択一的に使うことができよう.

解釈実行時に記号表が使えるようになっていると, ほかのデバッグ用の道具を提供するのが容易になる. 例えば, 標準手続き DUMP を呼んで先に述べたように記号表を文字でダンプしてから実行を再開するようにすることができる. また, PROCEDURE DUMP を呼出して, 現在実行中の手続き用の記号表の一部だけをダンプすることもできるだろう.

重要であってしかも作るのが容易な道具は“トレース”である. TRACE (A) を実行すると, その時点から, A に値が代入されるたびに, 新しく代入された値を印刷する. NOTRACE (A) を実行するとその機能を打消す.

これを実現するには, 記号表のエントリに1ビットの欄を設けて, TRACE

(A) (NOTRACE (A)) が記号表の A のエントリのこのビットを 1(0) にする. CASE 文の := 演算子に対する部分実行文の中で，このビットを検査してもし1であれば PRINTVALUE ルーチンを呼び出すようにするのは簡単な仕事である.

同様に，名札のトレースでは，その名札に分岐が生ずるたびに，メッセージを印刷するようにできよう. これはインタプリタの CASE 文の分岐演算子 BRL に対応する部分実行文の中に実現できる. つぎのような変った方法もある：LABEL (L, A, B, ⋯, C) を実行すると名札 L に分岐が行われるたびに A, B, ⋯, C の現在の値を印刷させる.

読者が，自分で新しくデバッグ用の道具を作り出すことは大歓迎である. インタプリタの主要な利点は記号表が利用できることであることを再び繰返す. 第二に. ある種のデバッグ用の特色を実現するコードはポーランド記法の二，三の演算子に対する部分実行文だけに限定することができ，解釈実行時の能率をほとんど落さないでよい.

議　論

代入文と分岐を主に論じてきた. これはインタプリタの中で行われていることをできるだけ単純明快に記述するためである. 読者は当然これを，このほかの演算にも拡張できるはずである. 拡張するときは，中間結果を全て保持するためにスタックを最大限利用する. 例えば，手続きを呼出すときには，かげのパラメタもおもての実引数も呼出す直前にスタックの上に全てのせる. 作業用の場所を，このためによそから持ってくる必要はない. 実際，ブロック構造と再帰的手続きを備えた ALGOL 型言語に対しては，このスタックで，8.9 節に述べた実行時のスタックとしての機能を容易に果させることができよう.

多くのプログラム用言語では，解釈実行時に記号表を省略したり記号表への参照を省略したりすることが可能である. 結局，機械語による目的コードを実行するときに省略が可能であれば，インタプリタでも省略は容易である. これによってそのための記憶場所を別の目的のために使えるかもしれないが，プログラムの診断情報の提供を容易にするという，インタプリタがコンパイラに対して持っている利点の一つを失うことになる.

ここで議論したものだけが，この仕事を行う方法ではない. 読者は，きっと，自分で選んだプログラム用言語に対して変換を取扱いスタックの上の別々の値を表現するもっとよい方法を発見することができるだろう.

第17章

コード生成

第1節では，目的コードの形式をいくつか述べ，それぞれの利点を概説する．コード生成を議論するのに本章を通じて使用する機械語の記述を与える．第2節では，種々の内部形式となったプログラムから機械語を生成する方法を説明する．この節では，議論はもっぱら単純変数のみからなる算術式だけに限定し，コード生成の進行や中間結果用変数に伴なう問題などをできるかぎり明快に説明することに努める．主記憶上の被演算子は絶対番地で（指標レジスタを使用しないで）参照できるものと仮定している．

第3節は，式に含まれる被演算子の機械番地を生成する方法を詳しく議論する．これには，添字付変数や，実行時の環境が ALGOL 風な場合の局所的変数や大域的変数などが含まれる．

第4節では，コード生成を算術式以外の他の構成要素に拡張する．大部分の拡張は明らかなので，比較的重要な点あいまいな点に限定して注意を述べる．

複雑な高水準言語用のコード生成における問題点の一つはコード生成ルーチンが大きくてかさ張るようになることである．そこで第5節では，生成されるべきコードに関する情報をいくつかの表に小さくつめ合せて部分的に解釈できるようにする方法をいくつか議論する．最後の第6節で IBM 360 オペレーティングシステムのもとでの目的モジュールの形式を述べる．読者があまり苦労せずに，この目的モジュールがどのようにして，そして何故このような形式をとるのかが理解できるようにしよう．

17.1 序　論

目的コードの形式

コードにはおよそ3通りの形式がある：第一は絶対形式で，絶対番地にお

かれ，計算機が直ちに実行できるものである．つぎはアセンブリ言語の形式のものでこれをアセンブルしなければ実行できない．最後は，機械語の形式であるがカードの形式で2次記憶におかれたものであり（バイナリ・デック，目的モジュールなどと称する），これを実行するためには，これ以外の副プログラムと結合して主記憶上にロードしなければならない．

最初のものは，時間の観点から最も効率のよいものである．このようなコンパイラの例として WATFOR, PUFFT (Rosen ほか (65) 参照), ALGOL W などがある．50行程度の短かい原始プログラムであれば，大型機で1～2秒でコンパイルを終了して実行に移ることができる．主な欠点は，副プログラムをいくつか予めコンパイルしておき，実行前にそれらと結合することが容易ではないことである．使用する副プログラムは全て同時にコンパイルしなければならない．効率を取れば，融通性を失う．プログラムを小さくしようという環境では，コンパイラは絶対コードを生成するのは当然であるが，デバッグは大変である．

アセンブリ言語を生成するのは，3通りのうちで最も安易である．何故なら0と1の組合せとしての命令を生成するような面倒をしないですむからである．記号を用いた命令をカードの形式で生成すればよい．マクロ呼出しも生成でき，マクロ定義は事前にアセンブリ言語で書いておく．例えば，浮動小数点形式の整数を固定小数点形式に変換するには3～4命令必要な機械があるが，これらの3～4命令を生成するかわりに，コンパイラがマクロ呼出し FIX（仮称）を生成して後でアセンブラがこれを展開すればよい．これはコンパイラの規模を小さくするのにも役立つ．

このような利点があるにもかかわらず，アセンブリ言語を生成するのは選択として最悪であるのが普通である．プログラムの翻訳過程に1段階余分な処理を加え，しかもこの余分な段階に要する時間がコンパイルに要すると同程度であることが多い．

実用に供されているコンパイラの大部分は，目的プログラムを目的モジュール，あるいはバイナリ・デックの形式にしている．これは，機械語をカード形式にしたものを連ねたもので，詳細は17.6節を参照されたい．目的モジュールには，このプログラムが呼出している他のプログラム（副プログラム）の文字形式による名前と，他のプログラムから呼ばれる可能性のあるこのプログラム自身の中にある入口の名前が含まれている．この目的プログラムは，これら他のプログラムと“結合”され，主記憶にのせられ実行される．

これによってより融通性が増し，多くのシステムでこれを標準的な手段として採用している．しかしながら，結合とロードはそれ自体時間のかかる処理過程であることは注意しなければならない．

コード生成を記述するために用いる機械

コード生成を説明するのに仮想的な機械語を与えよう．この機械は**累算器**（**acc** と略す）を 1 個持ち，全ての算術演算はここで行われる．さらに指標レジスタを 7 個有している．この機械はコード生成の説明を容易に行える程度に単純化し，現実に起りうる問題の大部分を示すことができる程度に複雑化してある．

下記に挙げる命令群は，平均的な計算機に見られる典型的なもので，主記憶から acc またはレジスタへの格納命令，acc またはレジスタの内容を主記憶にしまう命令，acc へ数を加える（から減ずる）命令などがある．この機械は数の表現として固定小数点と浮動小数点の両形式を持つ．精しい内部形式はここではどうでもよい．指標レジスタ上の値は固定小数点形式である．

命令の内部で主記憶の番地を参照するために非負整数 k（オフセットあるいは変位）と 2 個の指標レジスタ，それに間接番地指定用に 1 ビットを用いている．したがって $k(i, j)$ によって参照される番地は

k ＋ レジスタ i の内容 ＋ レジスタ j の内容

である．もし i（あるいは j）がゼロならば，“レジスタの内容”は 0 を意味する．云いかえれば $k(0, 0)$ は番地 k を参照する．星印によって間接番地指定ビットが 1 であることを示す．すなわち $*k(i, j)$ によって $k(i, j)$ 番地の内容の示す場所を参照する．図 17.1 命令一覧表において，M はいま述べたように主記憶の参照を示す．

命令		意味
LOAD	M	M番地の内容をacc に置く。
LREG	i, M	M番地の内容をレジスタ i に置く。
STORE	M	acc の内容をM番地にしまう。
SREG	i, M	レジスタ i の内容をM番地にしまう。
LACCR	i	レジスタ i の内容をacc に置く。
LRACC	i	acc の内容をレジスタ i に置く。
ADD(ADDF)	M	M番地の内容をacc に加える。
SUB(SUBF)	M	M番地の内容をacc から減ずる。
MULT(MULTF)	M	acc の内容にM番地の内容を乗ずる。
DIV(DIVF)	M	acc の内容をM番地の内容で除す。DIV の場合，結果は端数を切捨てた整数。
FIX		acc の内容を浮動小数点形式から固定小数点形式に変換.
FLOAT		acc の内容を固定小数点形式から浮動小数点形式に変換.
ABS		acc の内容の符号を正にする。
CHS		acc の内容の符号を逆転する。
B	M	M番地の命令へ分岐する。
BM	M	acc の内容が負ならばM番地の命令へ分岐する（同様にBP, BZ なども ある）

ADD,SUB,MULT,DIV は固定小数点演算用
ADDF,SUBF,MULTF,DIVF は浮動小数点演算用

図 17.1 記号命令一覧表

17.2 簡単な算術式のコード生成

コード生成ルーチンは，各変数や中間結果用変数の記述（記号表などに置かれている）を使用する．この記述には，変数の型，実行時の番地（と等価なもの）その他必要な情報が含まれる．しかしながら，この節の目的はコード生成の基本的な考え方を概説し，内部形式プログラムの違いによって事態がどう変るかを示すことにある．簡単のため，この節では各変数はその文字のまま表し（記号表のエントリへのポインタではなく），アンセブリ言語によるコードを生成すると仮定しよう．手続き GEN(X, Y) を呼べば，演算 X，被演算子 Y のアセンブリ言語命令が生成される．ただし X と Y は STRING 型変数である．例えば，GEN('ADD', 'GAB').

ここで取扱う算術式は，演算子 ＋，－，*，／ と単項演算子としての － で，被演算子は全て整数型の単純変数だけに限定する．

(17.2.1)　A * ((A * B + C) − C * D)

を例として，4-組，3-組，ポーランド記法による内部形式からコードを生成することを説明しよう．

4-組からのコード生成

4-組による表記法では，演算は実行される順序に現れる．したがって，これを順にたどって，つぎつぎとコード生成すればよい．主要な問題は，acc の内容を跡付けることで，acc では実行時の全ての演算が行われるので，不必要な LOAD や STORE が実行時に行われないようにする．このため，STRING 型の大域変数 ACC を用意する．ACC は**コンパイル時**の変数で，いま生成したばかりの命令が**実行時**に演算を行った直後の acc 状態を示すものである．もし ACC に文字列 ′　′ が代入されていれば，acc が実行時に空であることを示す．ACC が空白文字でなければ，ACC には（いま生成された命令が実行時に実行された直後に）acc 上にある変数または中間結果用変数の名が入っている．

手続き GETINACC(X, Y) を呼び出すと，実行時の変数 X あるいは Y のいずれかを acc にのせる命令を生成する．例えば，乗算 X＊Y のコードを生成するには，GETINACC を呼出して，いずれかの被演算子を acc にのせればよい．X／Y のような演算は機械上で可換ではないので，最初の被演算子を acc にのせる必要がある．このような場合には，GETINACC(X, ′　′) と呼べば，X を acc にのせるコードを生成するように約束しておく．なお，GETINACC は，もし acc が現に空でなければ，acc の内容を退避する命令も生成しなけれ

PROCEDURE GETINACC(X, Y); 　STRING X, Y;	X または Y を acc にのせる命令を生成する (Y = '　' ならば X を acc に).
BEGIN STRING T;	T は中間結果用変数.
IF ACC = '　' THEN 　　　BEGIN GEN('LOAD',X); 　　　　　ACC:= X; RETURN 　　　END;	acc が空であれば LOAD X を生成し，大域変数 ACC を X と設定しておく.
IF ACC = Y THEN 　　　BEGIN T:= X; X:= Y; Y:= T END	もし Y がすでに acc にのっていれば，被演算子を交換する.
ELSE IF ACC ≠ X THEN 　　　　BEGIN GEN('STORE', ACC); 　　　　　GEN('LOAD', X); ACC:= X 　END;　END	もし X が acc にのっていなければ，acc を退避して X を acc にのせる命令を生成する.

ばならない場合があることに注意しよう.

前述したように,4-組を順にたどってつぎつぎとコードを生成する.順にたどるためにカウンタを用い,4-組の各欄をそれぞれ QD(*i*). OP, QD(*i*). OPER1, QD(*i*). OPER2, QD(*i*). RESULT で参照することにしよう.各演算子 +, −, *, / および単項演算子 − 用のコードを生成するためのルーチンを下記に示す.

最初のコード生成ルーチンは可換演算子 + 用で,第2被演算子がすでに acc にのっていれば,GETINACC ルーチンが *i* 番目の4-組中にある2個の被演算

+4-組用のコード生成

GETINACC(QD(i). OPER1, QD(i). OPER2); GEN('ADD', QD(i). OPER2); ACC:=QD(i). RESULT	acc に被演算子をのせるコードの生成. ADD 命令の生成. acc 状態を保持.

*4-組用のコード生成

GETINACC(QD(i). OPER1, QD(i). OPER2); GET('MULT', QD(i). OPER2); ACC:=QD(i). RESULT	被演算子を acc に. MULT 命令の生成. acc 状態の保持.

−4-組用のコード生成

GETINACC(QD(i). OPER1, ' '); GEN('SUB', QD(i). OPER2); ACC:=QD(i). RESULT	SUB では第1被演算子が acc 上になければならない. SUB 命令の生成. acc 状態の保持.

/4-組用のコード生成

GETINACC(QD(i). OPER1, ' '); GET('DIV', QD(i). OPER2); ACC:=QD(i). RESULT	第1被演算子が acc 上になければならない. DIV 命令の生成. acc 状態の保持.

単項演算子− 4-組用のコード生成

GETINACC(QD(i). OPER1, ' '); GEN('CHS', ' '); ACC:=QD(i). RESULT	被演算子はただ一つ. "符号反転" 命令の生成. acc 状態の保持.

4-組	生成された命令	ACC
(＊ A, B, T1)	LOAD A MULT B	T1
(＋ T1 C, T2)	ADD C	T2
(＊ C, D, T3)	STORE T2 LOAD C MULT D	T3
(－ T2, T3, T4)	STORE T3 LOAD T2 SUB T3	T4
(＊ A, T4, T5)	MULT A	T5

図 17.2　4-組からのコード生成

子を実際に切換えていることに注意しよう．最後のコード生成ルーチンには単項演算子一用で，ここでは，被演算子はただ1個である．したがって GETINACC 呼出しの第2引数には′　′を用いる．

図17.2で第1欄は，算術式（17.2.1）に対する4-組の系列，第2欄は，それぞれの4-組に対応した生成コード，第3欄は，このコードが生成された直後の大域変数 ACC の内容をそれぞれ示す．中間結果用変数 T_i はそれぞれコンパイル時の記述を用いているけれども，その変数が活きている範囲内でそのまま acc 上に残されている中間結果に対しては，記憶場所を割付ける必要がないことに注意.

いま設計したばかりのコード生成ルーチンは，共通式を括り出したような算術式に対しては正しく働かないことがある．例えば，算術式 A＊B＋A＊B に対するコードを生成するために先のルーチン群を読者自ら実行してみて誤りを発見することを奨める．ただし，括り出しを済ませた4-組は(＊A, B, T1)(＋T1, T1, T2)である．これに対する修正は次節で扱う.

3-組からのコード生成

4-組の場合の主な欠点は，全ての中間結果に関する記述をコンパイル時を通じて保持しておかねばならないことである．3-組の場合は，これは不要で，その上，3-組には一つあたり3欄しか必要ないのでプログラムの内部形式の規模がより小さくてすむ.

3-組を使用する時には，中間結果用の記述が必要とはなるが，それを参照するコードを生成している間だけその記述を保持しておけばよい．例えば，3-組

(10) (* A, B)

に対するコードを生成する際に，この結果用の記述が生成される．3-組10を最後に参照する3-組に対するコードを生成した後には，この結果用の記述は削除してよい．

中間結果用変数の活きている範囲が，互に共通部分を持たないか，入れ子になっている（14.2節参照）ならば，中間結果用変数の記述を保持するためにコンパイル時のスタックを使用することができる．さもなければ，これらの変数に場所を割付けたり取り上げたりするためにはもっと複雑な機構が必要となる．ここでは，前者の場合のコード生成法を説明しよう．さらに，各中間結果はただ一度しか参照されないと仮定する（これは，算術式を最適化しない通常の場合である）．

コードの生成の間，スタック TRIP を設けて，コードは生成されたが，その結果の値を参照するコードはまだ生成されていない 3-組 の番号 をたくわえる．このスタックと並列のスタック TEMP には，この結果の値を代入する中間結果用変数の名前 T_k が入る．TRIP と TEMP の現在の要素の数をカウンタ j で示す．

3-組 i に対するコードを生成する前に，2個の被演算子を検査して，いずれか一方がそれ以前にある3-組を参照していれば，その3-組に対応する中間結果の名をその被演算子と置き換えなければならない．これは，下に示す手続き FIXTEMP(X, Y) を呼出すことによって行う．X は，検査すべき被演算子の欄を示し，Y には，手続きから戻ってくる際に変数か中間結果の名が入っている．

3-組 i に対するコードが生成されると，そのコードを実行した結果得られる値を記述する名を生成しなければならない．この名と3-組の番号をそれぞれス

PROCEDURE FIXTEMP(X, Y)

IF X が3-組 kを参照している THEN	もし X が 3-組k を参照していれば，ス
BEGIN FOR m:=j STEP −1 UNTIL 1 DO	タック中に k を探し，それに対応する
IF TRIP(m) = k THEN GO TO F;	TEMP にある名前を Y に得る．
F: Y:=TEMP(m)	
END	さもなければ，X は変数名であり，こ
ELSE Y:=X;	れを Y に置く．

PROCEDURE NEWTEMP

T:=新しい中間結果の名; j:=j+1; TEMP(j):=T; TRIP(j):=i; ACC:=T;	新しく中間結果の名を生成する. これをTEMPスタックにのせ，TRIPスタックには，3-組番号iをのせる. (iは大域的な変数). acc状態の変更.

タック TEMP と TRIP にのせる．今生成したコードを実行するとその結果は acc 上にあるので，acc の状態を示す ACC も変更しなければならない．手続き NEWTEMP がこの仕事を行う．

+3-組用コード生成(*3-組のコード生成も同様)

FIXTEMP(TR(i). OPER1, T1); FIXTEMP(TR(i). OPER2, T2); GETINACC(T1, T2); GEN('ADD', T2); IF TR(i). OPER1が3-組を参照している THEN j:=j−1; IF TR(i). OPER2 が 3-組を参照している THEN j:=j−1; NEWTEMP	被演算子の名をそれぞれT1とT2へ置く. ADD命令の生成. もし被演算子が3-組を参照していれば， その名をスタックから削除する. この名はもはや必要ない. acc状態を更新.

−3-組用コード生成(/3-組のコード生成も同様)

FIXTEMP(TR(i). OPER1, T1); FIXTEMP(TR(i). OPER2, T2); GETINACC(T1,' '); GEN('SUB', T2); IF TR(i). OPER1 が 3-組を参照している THEN j:=j−1; IF TR(i). OPER2 が 3-組を参照している THEN j:=j−1; NEWTEMP	この手続きは，本質的には+のものと同じ．唯一の違いは，−は可換でないので，第1被演算子がacc上にあることを確かめてからSUB命令を生成しなければならない点である.

単項演算子−3-組のコード生成

FIXTEMP(TR(i). OPER1, T1); GETINACC(T1, ' '); GEN('CHS', ' '); IF TR(i). OPER1 が 3-組を参照している THEN j:=j−1; NEWTEMP	被演算子の名前をT1に置く. 命令の生成. 被演算子が中間結果であれば，それをスタックから削除. acc状態の変更.

コードを生成するには，カウンタ i を用いて3-組をたどる，i 番目の3-組は TR(i).OP, TR(i).OPER1, TR(i).OPER2 として参照すると仮定する．各演算子用コード生成手続きは4-組で用いたものと同様である．違いとしては，3-組の番号に対応する名を見付けたり，スタックから中間結果用の名を削除したり，各3-組の結果名をスタックにのせたりすることがある．

2個の中間結果用変数がそれぞれ活きている範囲は共通部分を持たないか入れ子になっていると仮定しておいたので，コードが生成されているその時々に参照すべき中間結果の名はつねにスタックの上から2個の要素のいずれかにある．したがってこれを探すには，スタックの中を全て見るのではなく上の2個の要素を探せばよい．これによって手続き FIXTEMP が簡単になる．

図17.3に算術式 (17.2.1) に対する3-組からのコードを生成する様子を示した．第1欄は3-組の系列，第2欄は，それぞれの3-組に対して，被演算子が別の3-組を参照している場合にそれに対応する中間結果用変数の名で置換えたもの，第4欄は3-組を処理した直後におけるスタック TRIP と TEMP の内容，第5欄は，同時点での acc の状態を示す．なお，生成されたコードは，4-組に対するものと同様である．

中間結果用変数の記述（すなわち名前）の生成と削除に関する注意を二，三

	3-組	置換え後	命令		スタック	ACC
(1)	(* A, B)	(* A, B)	LOAD	A		
			MULT	B	1, T1	T1
(2)	(+ (1) , C)	(+ T1, C)	ADD	C	2, T2	T2
(3)	(* C, D)	(* C, D)	STORE	T2		
			LOAD	C	3, T3	
			MULT	D	2, T2	T3.
(4)	(− (2) , (3))	(− T2, T3)	STORE	T3		
			LOAD	T2		
			SUB	T3	4, T4	T4
(5)	(* A, (4))	(* A, T4)	MULT	A	5, T5	T5

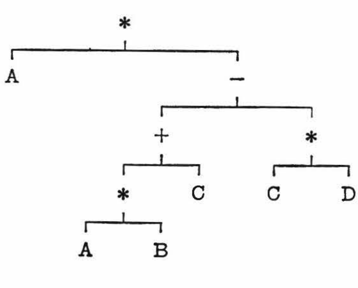
図 17.3 3-組からのコード生成

述べよう．コードがたくさん生成されてしまうと，2度と処理されないような3-組ならば，3-組の結果名で3-組を置換えてしまうことができる．すなわち，3-組 i に対するコードが生成されると，この結果に割付けられた名前を TR(i) の一つ（あるいはいくつか）の欄に置くことができる．したがって，この場合はスタックが不要となり，名前の削除も必要がなくなる．

もし，中間結果の値が，何度も参照される可能性があり，その活きている範囲が共通部分を持っていたり，入れ子になっていなかったりすると，活きている範囲がどこで終っているかを知る必要がある．これは中間結果の記述を削除するためばかりでなく，実行時のデータがいつ使用されるかを知り，それをしまうためのコードを生成するためにも必要である．これについては，後で詳述しよう．

木からのコード生成

11.4節で説明したように，単一の算術式に対する n 個の3-組の系列は，木とみなすことができる．ここで TR(n) が木の根を表している．図17.4は，算術式（17.2.1）に対する3-組とそれに対応する木を示している．さて，ここでは，木の根に当る節点（最後の3-組が最初になる）からコードを生成しよう．これによってコードを生成する順序に自由度が増し，（17.2.1）に対してこれまでよりも良いコードが生成できるかもしれない．再帰的手続き COMP(i) を使用できるようにする．この手続きの目的は根が TR(i) である部分木に対するコードをコンパイルするものである．COMP(i) の動作は i 番目の3-組の演算子 TR(i).OP と被演算子 TR(i).OPER1 と TR(i).OPER2 とに依存する．それぞれの演算子に対する操作を398ページの行列に要約してある．＋



(1)　(*A,　B)

(2)　(+(1),　C)

(3)　(*C,　D)

(4)　(−(2), (3))

(5)　(*A,　(4))

図 17.4　(17.2.1) の木表現

演算子用の表を考えよう．もし TR(i).OPER1 と TR(i).OPER2 がともに変数名であれば，OPER1 を acc にのせ，OPER2 を acc に加える2命令を生成する．もし OPER1 が変数，OPER2 が部分木で，その根として演算子を持っていれば，TR(i).OPER2 が3-組のリスト TR 中のその根への指標となっている．その部分木に対するコードを生成するために COMP(i) を再帰的に呼び出す．この部分木に対する式の値を acc にのせるコードが生成されて，COMP から戻ってくると OPER1 を被演算子として ADD 命令を生成する．

もし被演算子がともに部分木でそれぞれの根が演算子になっているならば，まず第1被演算子に対するコードをコンパイルするために COMP を呼出す．そしてそれに対する値が acc にのったことを示すように被演算子 TR(i).OPER1 を変更する．操作 REPEAT は，なすべき仕事を決めるために再度先ほどの行列を用いることを意味する．ここでは TR(i).OPER1 = ACC と TR(i).OPER2 = 部分木とによって規定される操作を実行する．このようにして，新しい中間結果名を生成し，acc の内容を退避する STORE 命令を生成し，第2被演算子に対するコードをコンパイルし，最後に ADD 命令を生成する．

下記の表は，図 17.4 の木に対して COMP(5) と呼び出した時に生成されたコードと一連の操作を示している．第1欄は処理中の節点，第2欄はその時点の節点の内容，第3，4，5欄はそれぞれコードが生成される前に実行される操作，生成されたコード，コード生成後の操作を示している．3-組 (−(2), (3)) では，第2被演算子のコードを先に生成することができたので，LOAD 命令と STORE 命令を省略することができた．

3-組		操作	コード	操作
(5)	(* A, (4))	COMP(4)		
(4)	(− (2), (3))	COMP(3)		
(3)	(* C, D)		LOAD C	
			MULT D	RETURN
(4)	(− (2), ACC)		STORE T1	COMP(2)
(2)	(+ (1), C)	COMP(1)		
(1)	(* A, B)		LOAD A	
			MULT B	RETURN
(2)			ADD C	RETURN
(4)			SUB T1	RETURN
(5)			MULT A	RETURN

＋に対する行列（＊に対する行列も同様）

OPER 2 / OPER 1	ACC	変　数	整数(部分木)
ACC	無　効	GEN ('ADD', TR(i). OPER2)	T:=新しい中間結果の名 GEN ('STORE', T) COMP (TR(i). OPER2) GEN ('ADD', T)
変　数	GEN ('ADD', TR(i). OPER1)	GEN ('LOAD', TR(i). OPER1) GEN ('ADD', TR(i). OPER2)	COMP (TR(i). OPER2) GEN ('ADD', TR(i). OPER1)
整　数 (部分木)	無　効	COMP (TR(i). OPER1) GEN ('ADD', TR(i). OPER2)	COMP (TR(i). OPER1) TR(i). OPER1:=ACC REPEAT

ーに対する行列（／に対する行列も同様）

OPER2 / OPER1	ACC	変　数	整数(部分木)
ACC	無　効	GEN ('SUB', TR(i). OPER2)	無　効
変　数	T:=新しい中間結果の名 GEN ('STORE', T) TR(i). OPER2:=T REPEAT	GEN ('LOAD', TR(i). OPER1) GEN ('SUB', TR(i). OPER2)	COMP (TR(i). OPER2) T:=新しい中間結果の名 GEN ('STORE', T) TR(i). OPER2:=T REPEAT
整　数 (部分木)	T:=新しい中間結果の名 GEN ('STORE', T) COMP(TR(i). OPER1) GEN ('SUB', T)	COMP (TR(i). OPER1) GEN ('SUB', TR(i). OPER2)	COMP (TR(i). OPER2) TR(i). OPER2:=ACC REPEAT

単項演算子ーに対する行列

OPER1	ACC	変　数	整数(部分木)
	GEN ('CHS', ' ')	GEN ('LOAD', TR(i). OPER1) GEN ('CHS', ' ')	COMP (TR(i). OPER1) GEN ('CHS', ' ')

ポーランド記号列からのコード生成

ポーランド記号列を解釈実行することについては第16章で議論した．解釈実行する代りに本質的に同様な方法でコードを生成することができる．左から右に演算子と被演算子をたどる．被演算子に出会うとその値をスタックの上にのせる代りに，その記述（ここでは簡単な機構なので名前そのもの）をスタックの上にのせる．（2項）演算子に出会うと，（2項）演算を行う代りに，被演算子を記述しているスタックの上から2個の要素にある記述を用いて，（2項）演算を行うコードを生成し，スタックには2個の記述に代って結果に対する記述をのせる．

2項演算子は，つねにスタックの上から2個の要素を使用するので，もしこのほかのスタック要素がacc上の値を記述していれば，この2項演算用のコードを生成する前にaccを退避する命令を生成しなければならないことに注意．これを見るとaccの内容を跡付ける方法が別にいることがわかる．

大域的変数 ACC は使用しない．その代り，スタック要素には 'ACC' という文字列が入って値がacc上にあることを示す．ポーランド記号列の中で被演算子名に出会うと，

1. スタックの上から2番目の要素が 'ACC' であれば，新しい中間結果名 T_i を生成し，STORE T_i 命令を生成し，そのスタック要素に T_i の名前を置く．
2. 被演算子名をスタックの上にのせる

という仕事を行う．したがってacc上の値が記述されているのはスタックの上から2個の要素以外にはない．2項演算子は決してaccの内容を退避すべきかどうかを調べるには及ばない．単項演算子は上から2番目の要素だけを調べてaccの内容を退避すべきかどうかを決めなければならない．

意味付けルーチンにおけるコード生成

もし意味付けルーチンが3-組や4-組を生成し，それからその順にコードが生成されるのなら，意味付けルーチンが，直接，そのコードを生成してしまうことができる．すなわち

原始プログラム → 4-組 → コード

の代りに

原始プログラム→コード

が得られる．もちろんこれによって意味付けルーチンがより複雑になるが，コンパイラは全体として，より速くなる．われわれはコードを生成中の被演算子について十分知っていなければならない．被演算子の記述は，正に完全でなければならない．これによって，名前は参照される前に必ず宣言されていなければならないなど何らかの制限を言語に加えることになるかもしれない．

より良いコードの生成

コード生成システムを作成する上で生ずる問題の一つに，目的コードの命令を適宜省略するために種々の工夫をしなければならないことがある．これらの工夫は，コードを生成する個々の計算機の機械語に依存しているので標準化することは難かしい．これらの工夫を実現するには，個々の演算子に対するコードを生成するプログラム片のおのおのが一つ，二つあるいはそれ以上の特別な場合を探してそれぞれに対して別々のコードを生成する必要がある．これによってコード生成システムが大きくそしてかさ張るようになる傾向がある．二，三の例を示せば十分だろう．

MOVE 命令を持った機械がある．この命令はある記憶場所の内容を他の場所に移すものである．したがって，もし4-組（:＝A,，B）に対するコードを生成することになって，A が acc 上になければ，

```
LOAD  A        よりはむしろ        MOVE A, B
STORE B
```

を生成すべきだろう．しかしながら，それにつづく4-組が，例えば，(＋A, C, T1) であれば，LOAD と STORE を生成する方が好ましい．何となれば，A の値がとにかくも必ず acc 上にあるからである．

番地部を直接加えたり，acc にのせたりする命令（“add immediate”, “load immediate” など）を持った機械もあり，加えたりのせたりする値は命令自体の一部である．したがって，4-組（＋A, 3, T1）に対しては

```
LOAD A        よりはむしろ        LOAD A
ADD  L3                           ADDI 3
```

を生成すべきであろう．ただし L3 は整数値3の入っている記憶場所である．

これによって記憶場所ばかりか，この命令系列を実行するたびに主記憶を参照する時間も節約することができる．

逆除算命令 INDIV（または逆減算命令）を持った機械もあり得る．ここでは，分母（または減数）が acc 上にのる．このような場合，4-組（/ A, B, T1）に対して，B がすでに acc 上にあれば

```
LOAD A      よりはむしろ      INDIV A
DIV  B
```

を生成すれば LOAD 命令と STORE 命令を省略できる．

LOADN（符号を逆転して acc にのせる命令）を使用すると得になることがある．例えば4-組（－A, , T1）に対して，A が acc 上になければ

```
LOAD A      のかわりに      LOADN A
CHS
```

を生成する．

ALGOL の ABS や ENTIER のような関数では，サブルーチンを呼出すよりは，コードを直接組込む方が応々にして好い場合がある．

17.3 被演算子の番地の決定

前節では，さまざまな内部形式プログラムからコードを生成することを概説した．事態が複雑にならないように，各被演算子は，英字名でそのまま表した．この節では，機械語命令の中で被演算子への参照を生成する方法を決定する問題を扱う．これは，ALGOL 型言語にとっては複雑な仕事で，それはこのような言語では，実行の時点に，別々のデータ領域を複数個参照したり，添字付変数が現れたりするからである．

算術式に対する処理過程を，被演算子を整数型のみに限って説明しよう．これは，問題点に注意を集中するためである．プログラムの内部形式は4-組とする．コード生成の一般論には変わりがないことに注目しよう．気を配らねばならないのはもっと細かい点である．

ここでは，8.9 節で議論したように手続き宣言が入れ子になることを許した ALGOL 型言語を仮定する．コード生成のどの点をとっても，1個の手続きの本体に対するコードを生成している．それに対応する実行時の1時点で，その手続きのデータ領域（DA と略す）と DISPLAY（**活きている**データ領域と活

```
BEGIN INTEGER N, B; PROCEDURE X;
 BEGIN INTEGER P;
  PROCEDURE Y;
   BEGIN INTEGER M;
         INTEGER ARRAY C[1:10];
          :
         … B+P+C[M]+C[N] …
END END END
```

主プログラムの DA の番地
手続き X の DA の番地
手続き Y の DA の番地
● ● ●

図 17.5　プログラム片と手続き Y の DISPLAY

きている DISPLAY）の番地はレジスタ1にあると仮定する（図17.5参照）．レジスタ1はこれ以外には使用しない．

添字付変数を参照するためのコードを生成するためには，ドープ・ベクトルの機構を仮定し，添字付変数を参照するのに（8.4.3）式を用いる．配列名に対する記号表のエントリにはドープ・ベクトルの対になった番地（データ領域番号，オフセット）が入っており，ドープ・ベクトルの第1語には，CONSPART が入っていると仮定する．

ここで，生成したいコードの型の例を示そう．図17.5のプログラム片を考える．実行時に，手続きYを実行すると，活きているデータ領域の番地（と DISPLAY，これは図の右に示す）は，レジスタ1にある．算術式 B＋P＋C[M]＋C[N] に対して生成したいコードは図17.6に示す．各変数を参照するオフセットは，読み易さを考えてその変数名によって与えてある．

最初に参照される被演算子はBとPなので，これらが属するデータ領域の番

LREG	2, 0(1)	B と P を参照するために主プログラムと X の DA の番地をそれぞれレジスタ2, 3 に得る.
LREG	3, 1(1)	
LOAD	B(2)	レジスタ2 を用いて，B を acc にのせる．2 には主プログラム DA の番地が入っている.
ADD	P(3)	P を加える，この変数は X の DA に含まれる.
LREG	4, M(1)	M をレジスタ4 に得る.
LREG	5, C(1)	配列Cの CONSPART をレジスタ5 に得る.
ADD	0(5, 4)	レジスタ4,5 には CONSPART＋M があるので，C[M] を (B＋P) に加える.
LREG	6, N(2)	N は主プログラム用 DA にあるのでレジスタ2 を用いる.
ADD	0(5, 6)	レジスタ5 には配列Cの CONSPART があり 6 には N があるので，C[N] を acc に加える.

図 17.6　B＋P＋C[M]＋C[N] に対するコード

地をそれぞれレジスタ2と3にのせるのが最初の2命令である．つぎの2命令はB＋Pを実行するもの．つぎに，C[M]を参照するため，Mをレジスタ4に置き配列CのCONSPARTの値をレジスタ5にのせる．7番目の命令はC[M]をB＋Pに加えるもの．つぎは，添字Nを参照するために，主プログラムのデータ領域の番地が必要となるが，これはすでにレジスタ2の上に得られている．同様に配列CのCONSPARTの値もレジスタ5に得られている．

解決すべき主要な問題の一つにレジスタを有効に使用するようなコードを生成することがある．上の例ではレジスタ2と5の内容は2度使われていたことに気を付けよう．ともかくも各レジスタに何が入っているかいつも憶えておかなければならない．

コンパイラの中には，レジスタにのせる必要のある値全てを別々に保持するレジスタ $R_1, R_2, \cdots, R_n$ が，必要な個数存在すると仮定してコードを生成するものがある．そのつぎのパスでは，いま生成されたコードを走査して，記号で書かれたこれら n 個のレジスタを現実の7個のレジスタに割付ける．この割付けを行いながら，実際のレジスタの間に使用上の衝突が起きないように注意してLREGやSREG命令をコードの間に挿入する．もちろんこのようにレジスタの割付をする目的は必要なLREGやSREGの数を最小にすることである．

われわれの用いている例ではレジスタの割付けはコードを生成しながら行う．これを能率よく行うためには，1組のレジスタ記述（後で詳述する）を設ける．これは前節で用いたACCの記述に非常によく似たものである．

これらのレジスタ記述の維持のためには時間がかかりコード生成法が一層複雑になる．良いコードを作ることより，高速で単純なコンパイラを作成することに関心があるならば，レジスタ記述は全く忘れてよい．任意の4-組からコード生成を開始した時点で，レジスタの内容は未定義（ただしレジスタ1を除く）と仮定し，レジスタ2, 3, $\cdots$, 7が必要なものとして割付ける．図17.6でB＋P＋C[M]＋C[N]に対するコードを生成する時にこれを行うならば，LREG命令をもう2個追加しなければならなかっただろう．

本節の内容は詳細に渉ることが多いので，込入った複雑な印象を与えるかもしれない．以下に本節で使用する変数と手続きの一覧表を示す．一覧表をすぐに憶えるには及ばない．多分まごつくだけである．掲げる前に取扱う内容を概観しておこう：

1. 4-組における実行時変数の記述の式.

2. レジスタ記述および記述の書式とそれらの用法．レジスタまたはacc上のデータ記述を OPERAND と称する．
3. 実行時データを参照するために生成された命令の一部となる実行時番地の形式．
4. レジスタに値をのせるコードを生成し命令の番地部を組立てる手続き群．
5. ＋4-組用のコード生成手順．

使用する変数と手続きの一覧表

1. QD(*i*).OP, QD(*i*).OPER1, QD(*i*).OPER2 および QD(*i*).RESULT. 4-組 *i* の4欄．最初を除く3欄の形式は図11.2と同じもの．
2. INS1 および INS2．ともに大域的変数．4-組に対するコードを生成する時，2項演算または単項演算の被演算子の機械語番地を保持するために使用する．
3. XRVALUE および XRSTATUS．ともに配列で，acc および各レジスタの現在の内容を記述するために使用．XRVALUE(0) が acc を記述し，XRVALUE(*i*) がレジスタ *i* を記述する．
4. PROCEDURE FREEACC. acc の内容を（必要ならば）退避するコードを生成し acc 記述を“空”に変更する手続．
5. PROCEDURE FREEREG(I). I ≠ 0 ならば，レジスタ I の内容を（必要ならば）退避するコードを生成しそのレジスタ記述を“空”に変更する手続き．I = 0 ならば，現在のレジスタ記述に基いて“空”にするレジスタを手続きが選択することができる．選ばれたレジスタの番号を引数 I に入れて戻る．
6. PROCEDURE GETINREG(OPERAND, I). OPERAND を適当なレジスタにのせ，そのレジスタ番号を I に入れて戻る．OPERAND には添字付変数を記述してはいけない．
7. PROCEDURE GETINACC(OP1, ADD1, OP2, ADD2). 前節にあったような手続きで，（必要ならば）OP1 か OP2 のいずれかを acc にのせるコードを生成する．OP*i* は被演算子，ADD*i* はそれに対応する機械語番地（オフセット表示，1～2個のレジスタと間接番地指定ビットを持つ）．OP2 がすでに acc 上にあれば，二つの被演算子を交換する．
8. PROCEDURE FIXAD(OPERAND, INSTR). INSTR に OPERAND を参

照するための番地（オフセット，1～2個のレジスタ，間接番地指定ビット）を置く．OPERANDがすでにacc上にあれば，0(0,0)を返し，レジスタj上にあればj(0,0)を返す．

9. PROCEDURE FIXADMEMORY (OPERAND, INSTR). この手続きはFIXADと同じでありこの設計は読者に任せる．FIXADとの唯一の違いはINSTRの中に主記憶を参照する番地を入れて戻らねばならぬことである．したがって，もしOPERANDがaccまたはレジスタ中にあり主記憶上に存在しなければ，OPERANDを主記憶上に退避するコードを生成し，退避した番地を返さなければならない．

4-組の被演算子

4-組の各被演算子用の欄と結果用の欄は11.1節の図11.2に描いたような形式になろう．特定の形式はここでは重要ではなく，情報はOPERANDに含まれている．添字付変数は，他に比べて複雑な形式になっていることに注意．これは，最も能率のよいコードを生成するために必要となる．

被演算子記述の内部でつぎのような部分がコード生成のために重要である：

1. データ型の欄（実数型，整数型，論理型，手続きなど）．この欄は，これに対応する実行時のデータがどんな形式で記憶装置上にあるかを示す．本節の被演算子は全て整数型と仮定している．
2. 番地欄．この変数が割付けられている記憶場所が属するデータ領域の番号とそのデータ領域内でのオフセット．ここでデータ領域番号は，活きているDISPLAY表でこのデータ領域の番地を参照するのに便利なオフセットに容易に変換できるものと仮定する．これについてはこれ以上言及しない．

中間結果用変数表の中の記述にはその中間結果が活きている範囲を示す欄が設けてある．この範囲を調べる3通りの方法については14.2節で概説した．ここでのコード生成機構はコードを生成しながら中間結果用に記憶場所の割付を行うもので，そのために別個のパスを設けるものではない．さらに，退避する必要のない中間結果に対しては（それが活きている範囲を通じてaccあるいはレジスタ上に置かれているならば）記憶場所の割付はしない．

ある時点でaccやレジスタの内容を退避しなければならないとすると，必要に応じて，その中間結果に記憶場所を割付けて，退避を行う命令を生成するル

ーチンが働く．4-組に対するコードを生成すると，その各被演算子を調べて中間結果の活きている範囲がこの4-組のところで終ったかどうかを見る．もし終っていれば，この中間結果に割付けられていた記憶場所は解放される（14.2節参照）．

レジスタ記述と acc 記述

配列 XRVALUE には，いま生成されたばかりの命令を実行時に実行した直後の acc および各レジスタの実行時の内容に関する記述が含まれている．XRVALUE(0) は acc を記述する．XRVALUE(i) はレジスタを記述する（ただし $i=1, \cdots, 7$）．XRVALUE(i) にはつぎのような情報が含まれる：

1. 図 11.2 のような形式の被演算子．これはその特定の被演算子が acc あるいはレジスタ上にあることを示す．
2. つぎのように記述された配列

5	0	配列名用記号表のエントリへのポインタ

に対する CONSPART の値がそのレジスタに入っていることを示す被演算子．

3. データ領域の番地がそのレジスタに入っていることを示す被演算子：

6	0	データ領域番号

XRVALUE(i) 中に現れることができる，上に述べたような値を OPERAND（の記述）と呼ぶ．

レジスタ上に何があるかを知るほかに，その値がそこにのっていることの重要度を知る必要がある．例えば，レジスタ 1 には活きているデータ領域の番地がのっているので，これは決してよそに退避すべきではない．上の配列と並列に動く配列 XRSTATUS を用いて，XRSTATUS(i) にはつぎのような値が設定される：

1. 0　レジスタ i あるいは acc が空．
2. 1　この値は主記憶上にもある．したがって，この値を別のところで使いたい場合でもレジスタを退避させる必要はない．

3. 2 この値は主記憶上にはない.
4. 3 このレジスタの内容は決して変更してはならない.

手続きの本体に対するコード生成を開始するにあたって, XRSTATUS(J), J = 0, 2, ···, 7, に0を代入し XRSTATUS(1) には3を代入する. そして XRVALUE(1) には

6	0	この手続きに対するデータ領域番号

を設定する.

このようなacc記述とレジスタ記述は一つの4-組に対してコードが生成されるたびに更新される. これに関わる重要な手続きは FREEACC と FREEREG (I)の二つである. 最初の手続きは以下のようにして acc 記述と XRVALUE (0)と XRSTATUS(0) とを検査しaccの内容を退避する必要があればその命令を生成する:

1. XRSTATUS(0) = 0 ならば, すぐ戻る (acc は空).
2. XRSTATUS(0) = 1 ならば, acc 上の値は主記憶上にもある. XRSTATUS(0) を0にして戻る.
3. XRSTATUS(0) = 2 ならば, acc 上の値は主記憶にはない. このシステム内で, このことは XRSTATUS(0) が中間結果を記述しているときにのみ生起するはずである. この値のために活きているデータ領域に場所を割付ける (オフセットを仮に k としよう). ″STORE k(1)″ 命令を生成する. XRSTATUS(0) を0にして戻る.
4. XRSTATUS(0) = 3 になるようなことがあってはならない.

FREEREG(I)も同様な仕事をする. I ≒ 0 ならば, 必要に応じて, レジスタIを退避する命令を生成し, そのレジスタ記述を“空”に更新する. I = 0 ならば, 手続きが解放したいレジスタを選択できて, そのレジスタ番号を引数 I に入れて戻る. この手続きが上の選択を行う場合, 第1候補はもちろん現在空であるレジスタであり, 第2候補は, その値がすでに主記憶上にあるレジスタとすべきであり (これらの場合は命令を生成する必要がない), 第3候補が XRSTATUS(I) = 2 であるレジスタとすべきである. 解放すべきレジスタが一たん決まると, 実行すべき各段は FREEACC と同様である.

FREEREG が選択できるレジスタには，もう一つ制限がある．しかし，これに関する説明は，もう少し後まわしにしなければならない．

被演算子の番地の形式

(＋A, B, T1) のような演算に対するコードを生成する時に，主記憶上にあるAおよびBを参照するために使用する番地の形式を知らなければならない．

下記に内部形式プログラムに現れる異なった被演算子の一覧表を掲げる．第2欄は，対応する被演算子を番地付け可能にするために生成される可能性のある命令を示し，第3欄はその被演算子を最終的に参照する命令の番地部と指標レジスタ部を示している．間接番地指定ビットもその被演算子が明記する場合に1と設定されよう．

最初の二つのエントリは実際には，記憶場所を参照するのではなくて，被演算子が acc あるいはレジスタ上にあることを示すために採用している方法である．これが可能なのはわれわれが採用している実行時の機構では，記憶場所の参照にはレジスタが少くとも1個必要だからである．

被演算子	命令	INSTR
1. acc 上		0(0, 0)
2. レジスタ j 上		j(0, 0)
3. 活きている DA 内，オフセット k の場所		k(1)
4. 別の DA 内，オフセット k の場所	LREG m, DA の番地	k(m)
5. 活きている DA 内の添字付変数 CONSPART のオフセットは k	LREG j, 添字 LREG n, k(1)	0(n, j)
6. 別の DA 内の添字付変数 CONSPART のオフセットは k	LREG j, 添字 LREG m, DA の番地 LREG n, k(m)	0(n, j)

手続き GETINREG (OPERAND, I)

上の一覧表をみると，添字や配列の CONSPART やデータ領域の番地をレジスタにのせる命令を絶えず生成しなければならないことがわかる．以下に示す手続きは，OPERAND に対してこの仕事を行い，I にレジスタの番号を置くも

のである．制限がただ一つあって OPERAND は添字付変数であってはならない．この手続きは局所的変数として i, J, D, OP, k を使用する．このルーチンには何ら技巧的な部分がないこと（ここで記述する他の大部分のルーチンも同様）を知ってほしい．これはまさしく細心の注意をもって OPEPAND を場合分けするものにほかならない

1. OPERAND がレジスタ j 上にあれば，j を I に置いて戻る（これは OPERAND を XRVALUE(j), $j=1, \cdots, 7$, と比較することによって調べる）．
2. OPERAND が acc 上にあれば，つぎのステップを行う：I:＝0; FREEREG(I) を呼び出す; "LRACC I" 命令を生成する．XRSTATUS(0) と XRVALUE(0) を XRSTATUS(I) と XRVALUE(I) に複写して戻る．
3. OPERAND が(6, 0, D)の形式ならば，D はデータ領域番号で，それに対応する番地をレジスタ上に得たい．つぎのステップを行う：I:＝0; FREEREG(I)を呼出す．データ領域 D の番地を参照するために k を活きているデータ領域内のオフセットとする．"LREG I k(1)" を生成する．XRSTATUS(I) に1を，XRVALUE(I) に OPERAND を代入して戻る．
4. OPERAND が定数か，中間結果か，変数名の場合それぞれの記述は適当な記号表にある．Dをそのデータ領域番号，k をそのオフセットとしよう．被演算子を OP＝(6, 0, D) に作り GETINREG(OP, J) を呼出す．（これによってデータ領域の番地をレジスタ J に得る．これは再帰的呼出しになっていることに注意.）I:＝0; FREEREG(I) の呼出し; OPERAND の間接番地指定ビットが0か1に応じて "LREG I, k(J)" または "LREG I, *k(J)" の生成; XRSTATUS(I) に1を, XRVALUE(I) に OPERAND を代入して戻る．

手続き FIXAD(OPERAND, INSTR)

ある4-組の OPERAND を与えると，この手続きはこれを参照する命令の番地部を作り上げ，その番地を INSTR に置く．INSTR に得られた結果も，OPERAND がすでに acc またはレジスタ上にあれば 0(0, 0) あるいは j(0, 0) の形式をとることがある．

1. OPERAND が acc 上にあれば，INSTR に 0 (0, 0) 番地を置いて戻る．

2. OPERAND がレジスタ j 上にあれば，INSTR に j(0,0) 番地を置いて戻る．
3. OPERAND が添字付であれば，つぎのステップを踏む：
   a）OPERAND の中で添字を記述する部分を SUBSCRIPT とする．GETINREG(SUBSCRIPT, I)を呼出す．INSTR に 0(0,I)番地を置く．
   b）配列名用記号表のエントリへのポインタをPとする．OP:＝(5,0,P)を作って，CONSPART をレジスタに置く命令を生成するために I:＝0; GETINREG(OP, I) を呼出す．番地部 0(I,0) と INSTR の論理和をとる．
   c）OPERAND で間接番地が指定してあれば INSTR の間接番地指定ビットを1にして，戻る．
4. OPERAND が定数か，中間結果か，単純変数で記号表のエントリへのポインタを持っている場合．値は acc 上にもレジスタ上にもない．つぎのステップを踏む：
   a）被演算子の番地のデータ領域番号をDとする．OP:＝(6,0,D) を作って，GETINREG(OP, I) を呼び出す．
   b）OPERAND のデータ領域内のオフセットを k とする．k(I) 番地を INSTR に置く．OPERAND で間接番地が指定してあれば，INSTR の間接指定ビットを1にして，戻る．

算術演算4-組に対するコード生成

ここでやっと i 番目の4-組(＋A, B, T1) に対するコード生成用の手続きの概要を述べる用意ができた．これ以外の4-組に対するコード生成も同様で，そのルーチンは読者に委せよう．最初の仕事は，2個の被演算子の番地を生成しそれぞれ大域的変数 INS1 と INS2 にのせることである(大域的変数を使用する理由は後述)．つぎは ADD 命令の生成に進む．このコード生成手順の概要は以下のとおり：

1. INS1:＝0;INS2:＝0 (被演算子番地を初期設定する)．
2. FIXAD(QD(i).OPER1, INS1) を呼出す．(これによって，必要に応じて第1被演算子を実行時に番地付けできるようにするためのコードを生成する．k(i, j) の形式をした実行番地を INS1 に置く)．

3. FIXAD(QD(*i*).OPER2, INS2) を呼出す.
4. GETINACC(QD(*i*).OPER1, INS1, QD(*i*).OPER2, INS2).(これは,被演算子の1個をaccにのせるために(必要があれば)命令を生成するもの.各被演算子は2個の値——OPERAND 記述自体とその被演算子の実行時番地——で与えられる.手続きから戻る際に,第1被演算子がacc上にある.)
5. このステップは,第2被演算子がaccまたはレジスタ上にあるときのみ実行する.目的は ADD 命令の生成であり,このためには被演算子が主記憶上になければならない.もし INS2 = 0(0, 0) か *j*(0, 0) であれば,主記憶番地を得るために,別の手続きを用いて番地計算を再度行わなければならない:INS2: = 0; CALL FIXADMEMORY (QD(*i*). OPER2, INS2).
6. INS2 番地を持った ADD 命令を生成する. QD(*i*).RESULT が acc 上にあることを示すように acc 記述を更新する(XRSTATUS(0) に 2 を代入).
7. もし2個の被演算子 QD(*i*).OPER1 と QD(*i*).OPER2 のいずれかが中間結果用変数でありその活きている範囲がこの4-組 *i* で終っているならば,つぎのように実行する:中間結果に記憶場所が割付けられていれば,それを解放する.中間結果がレジスタ上にあれば,レジスタ記述を"空"にする.
8. INS1: = 0; INS2: = 0.

INS1 と INS2 は大域的変数で,2項演算あるいは単項演算用のコード生成を行うプログラム全体で使用すべきものである.これはつぎの理由による.第1被演算子の番地を生成し,その番地がレジスタ2と3を使用しているとしよう.すると,第2被演算子の番地を生成する時に,レジスタ2と3には触れずにおくことが重要である.それは,これらのレジスタは第1被演算子を参照するのに必要な値が入っているからである.したがって,FREEREG はこれらのレジスタをそのままにしておかなければならない.このルーチンは先に述べたが,これにつぎのような機能を追加する:

> もし INS1 か INS2 のいずれかがレジスタ j を j (0, 0) か, k (j, m) か, k (m, j) の形で現在参照していれば,このレジスタを解放してはならない.

議　　論

これまでの議論はかなり詳細に渉ったように見えるかもしれないが，ここでしたことは，実際は個々の手続きの概要を述べただけにすぎない．例えば，どのレジスタに値が入っているかを決定する仕事が，できるだけ能率よく実行できるようにレジスタ記述を作成するよう努める必要がある．これは，各記号表のエントリに値がレジスタ上にあるかないかを示す欄を設けることになるかもしれない．あるいは，レジスタの状態に応じて，レジスタをいくつかの1次元リストに連結しておく必要があるのかもしれない（あるリストは空のレジスタ，あるリストは XRSTATUS(i) = 1 のもの，など）．

4-組からコードを生成する考え方は非常に簡単であるが，能率のよいコードを，たとえ局所的であっても，生成するとなるとこれは極めて複雑になる可能性がある．個々の手続きはいくつかの特例を探してそれぞれ個別的な処理を行うことから成っている．デバッグは決して終りにならず，それは，終ったと思うといつもあと一つ特例を忘れていたということになるからである．配慮すべき特例のいくつかとしては，(1)加算の二つの被演算子が同時に acc 上にあること（A＊B＋A＊B を最適化すると生ずる）(2)同じ値が同時に acc 上とレジスタ上にあること（M＋A[M] の加算で生ずる）があった．

明らかなことは，被演算子の番地を生成する過程でさえ，プログラムを作る前に，きちんと考えて，机上で実験してみるということである．

17.4　他種類の4-組に対するコード生成の拡張

この節ではまず算術式に対するコード生成に対してさらにいくつかの注意を述べる．つぎに，他種類の4-組に対するコード生成について述べる．考え方の基本は同じで，各4-組に対してどのようなコードを生成したいのかを知り，単純に，そのコードを生成する手続きを設計する問題である．したがって，ここでは，二，三の4-組演算子についてだけコード生成を手短かに述べ，時々見落しがちな点にいくつか触れるだけにとどめよう．

単項－演算子と ABS 演算子の最適化

よく行われている局所的な最適化技法はこれらの単項演算子をできるかぎり外側に出してこれらを削除しようとするものである．したがって算術式 －C＊(－(A＋B)＋D)を C＊(A＋B－D) に変えるには

－C＊(－(A＋B)＋D)→－C＊(－((A＋B)－D))→－－C＊
(A＋B－D)→C＊(A＋B－D)

の3段階を踏む．これによって，単項演算を二つ除去できた．

これをコード生成時に行うには，各中間結果の記述の中に2ビットの欄を設けて，その値でどんな演算をまだ施さなければならぬのかを示す：0＝＋（なにもしない），1＝－，2＝ABS，3＝－ABS．4-組(－A,,T)や(ABS A,,T)に対するコード生成手続きではコードは生成せずに，Aの状況に応じてTの2ビット欄に値を設定する．これは下の表に従う：

		中間結果Aの2ビット欄				原始プログラム
		+	−	ABS	−ABS	変数 A
(− A,,T)に対するTの2ビット欄	T:	−	+	−ABS	ABS	−
(ABS A,,T)に対するTの2ビット欄	T:	ABS	ABS	ABS	ABS	ABS

さらに，Aがaccあるいはレジスタ上にない場合，Tの記述子中にその値がAと同じであることを示すための用意をいくつか整えておかねばならない．

つぎに4-組(＊T1,T2,T3)に対するコード生成を説明しよう．ただし前述の2ビット欄は各被演算子に設けてある．T1とT2は中間結果と仮定する．もし原始プログラムの変数であれば，＋符号を自動的につけることにする．他の演算子 ＋，－，／ についての拡張は読者に委せよう．

4-組(＊T1,T2,T3)に対するコード生成手続きは図17.7に示す行列を眺めてT3の符号欄（2ビット欄）を決定する．行列要素が空白でなければ，コード生成手続きは，通常の乗算命令を生成して，行列要素にある符号をT3の符号欄に置くだけでよい．もし行列要素が空白であれば，T1(またはT2)に指定したABS関数を施し，T1(またはT2)の記憶場所（があればそこ）にその値を退避するコードを生成すべきである．対応する記述は更新されて，コード生成手続きは，変換された4-組に対するコード生成を再開する．

T1の符号欄 ＼ T2の符号欄	+	−	ABS	−ABS
+	+	−		
−	−	+		
ABS			ABS	−ABS
−ABS			−ABS	ABS

図17.7 (＊T1,T2,T3)におけるT3の符号欄の決定

単項演算を後に延ばすほどよいコードができるとばかりはいえない．符号欄が＋以外の中間結果がレジスタ上にあってこれを退避しなければならないならば，退避の前に単項演算を済ませた方がよかろう．

例えば，算術式 ABS(A＋B)＋C＊D に対する 4-組の系列（下記の第1欄）を考えよう．第2欄は ABS 演算をできるかぎり後回しにするコード生成である．第3欄は直ちに ABS 演算を行うコードを生成するもので第2欄より2命令少なくなっていることに注目しよう．

4-組	単項演算を延期した場合	通常のコード生成
(＋A, B, T1)	LOAD　A	LOAD　A
	ADD　B	ADD　B
(ABS T1, , T2)		ABS
(＊C, D, T3)	STORE T2	STORE T2
	LOAD　C	LOAD　C
	MULT　D	MULT　D
(＋T2, T3, T4)	STORE T3	ADD　T2
	LOAD　T2	
	ABS	
	ADD　T3	

型の混在した算術式に対するコード生成

4-組を生成した意味付けルーチンが必要な場所に型変換演算子を挿入したと仮定しよう．例えば K:＝A＊I＋J に対する4-組は下記の左に示すようなものになろう（A は実数型，I, J, K は整数型とする）：

(CVIR　I, , T1)	(＊　A, I, T1)
(＊R　A, T1, T2)	(＋　T1, J, T2)
(CVIR　J, , T3)	(:=　T2, , J)
(+R　T2, T3, T4)	
(CVRI　T4, , T5)	
(:=I　T5, , J)	

個々の演算子に対するコード生成ルーチンは，その被演算子と結果の型を確かに知っている．したがって＊R 用の生成ルーチンは，被演算子や結果の型を検査しないで MULTF 命令を生成し，＊I 生成ルーチンは，MULT 命令を生成し，

CVIR 生成ルーチンは FLOAT 命令を生成する．これは，コード生成ルーチンは個々にたくさんの部分ルーチンから成っているが，各々はかなり単純であることを意味している．

一方，4-組が上図の右側のようになっていれば，演算子 ＋，*，などに対するそれぞれの生成ルーチンは，被演算子と結果の型を検査して，必要ならば，型変換を生成し，どの演算を生成すべきか決定しなければならない．これらの仕事は，もとの意味付けルーチンで行っていたものを後まわしにして生成ルーチンに持ってきたものである．これは，おそらくもっと早くもとの意味付けルーチンの中で行うのが最善だろう．何となれば，コード生成部はすでに相当入組んでいるからである．

4-組 (:= A, , B) に対するコード生成

生成されるコードは，LOAD A, STORE B である．acc 記述は更新されて B が acc 上にあることを示す (XRSTATUS (0) に 1 を代入)．これによって，例えば，FORTRAN で

```
B = ···
IF(B)2, 3, 2
```

のような場合に，コードを短くすることができる．IF 文に対するコードの生成で，B がすでに acc 上にあることを知っているので，それを acc にのせる命令を生成する必要はない．

これと似たような状況には注意が必要である．代入文

```
B:= ··· + B + ···
```

でBのもとの値がレジスタ 2 に入っていて最後にBへの代入を実行する．代入のコードを生成した後には，レジスタ 2 の記述には“空”を設定しなければならない．すなわちこのレジスタにはBのもとの値が入っており新しいものではないからである．

(+ A, B, T1) のような4-組でも T1 への代入があることに注意しよう．違う点は，このような中間結果用変数への代入は 1 回限りで，したがって，この時点ではレジスタ上にはあるはずがない．2 ケ所以上の場所で値を代入される中間結果用変数は多少異なった取り扱いをしなければならない．

分岐と条件文に対するコード生成

分岐（BRL L）や（BR k）に対するコード生成には何ら問題はない．名札や分岐先の4-組に対する処理が済んでなければ，そこへの分岐を全て13.3節で4-組を生成する時に行った方法で1次元リストに連結しておく．

本章でこれまで扱ってきたコード生成は基本ブロックに対するもので，その中に分岐を含まず，よそからその中に分岐してくるようなものも含んでいない一直線の系列であった．このために各レジスタの中にあるものが何であるか正確にわかっていた．しかしながら，よそからの分岐がある4-組（または機械命令）においては，その4-組に対するコードを生成する直前の，accやレジスタ上にあるものが不明である．したがって，ここでは，全てのレジスタ等の記述は“空”と設定しなければならない（ただし XRSTATUS(J) = 3 のレジスタは除く）．レジスタを“空”にする場合は慎重を期す必要がある．もし記述が XRSTATUS(J) = 2 ならば，対応する値が主記憶にないことを意味するので，この値を退避する命令を生成しなければならない．

分岐される側の4-組に対するこのような検査を実現する一つの方法は，各4-組に余分な欄を設けて，分岐先になっているかいないかを示すために用いる．この欄は4-組を生成するパスで設定されよう．

条件文に対しては，レジスタの何らかの最適化を行うことができる．ここでは分岐がどのように生ずるか精しく知っているからである．条件文

IF ⟨E⟩ THEN S^1 ELSE S^2

に対して，S^1 や S^2 に対するコードの直前ではaccやレジスタの内容は同じである．このような条件文に対してつぎのような4-組を生成するとしよう（13.2節参照）：

(1)	T:=⟨E⟩ に対する4-組		(1)	T:=⟨E⟩ 対する4-組
(p)	(BZ $q+1$, T)		(p)	(IFTEST q, T)
($p+1$)	S^1 に対する4-組の系列	の代	($p+1$)	S^1 に対する4-組の系列
(q)	(BR r)	りに	(q)	(ELSE r)
($q+1$)	S^2 に対する4-組の系列		($q+1$)	S^2 に対する4-組の系列
(r)			(r)	(ENDELSE q)

すると，これに対するコードを生成する際に，新しい演算子IFTEST, ELSE, ENDELSEがあるために実行文の個々の部分がどこにあるか知ることができる．IFTEST 4-組に対するコード生成ルーチンには，T が偽の場合の分岐命

令を生成した後，現在のレジスタ記述 XRVALUE と XRSTATUS を退避しておいて，ELSE 4-組の中の使用していない欄の一つに，これらの値を指すポインタPを置いておく．つぎに，ELSE 4-組に対するコード生成ルーチンの中で，コードを生成した後に，先に退避しておいたレジスタ記述（P が指している）を大域的な配列 XRSTATUS と XRVALUE に復原する．こうして，S^1 と S^2 の直前ではレジスタの内容が同じであるという知識を利用することができる．

S^1 と S^2 に対するコードを生成した後のレジスタ記述をそれぞれ R^1, R^2 とするとこれらも退避しておいて，条件文を完全に実行し終った後のレジスタの状態を決定するために使うことができる．すなわち，各レジスタ i に対して，R^1 中にあるレジスタ i の記述が R^2 中にあるそれと同じであれば，以後につづく4-組に対するコードを生成するための記述としてそのまま用いる．もし等しくなければ，レジスタ i の記述は“空”と設定する．何となればそこになにが入るか不明だからである．

17.5 コード生成の整理統合

個々のコード生成ルーチンを集めると非常に大きくかさ張るものになることが多い．個々の4-組演算子に対して，一つづつ対応しているのであまりにもプログラム片が多くなる．しかしながら，これらのプログラム片の多くは極めて似通ったものが多く，GETINACC, GETINREG, FREEACC, GEN などのように，異なった操作の数はほんのわずかにすぎないことも事実である．したがって，場所を節約する方法を編み出すことが可能だろう．本節では，これらのいくつかを簡単に述べよう．

解釈ルーチン

場所を節約する方法の一つは，小さな言語を設計して，個々の4-組に対するコード生成ルーチンはこの言語で書き，この言語に対するインタプリタを作成するものである．この言語は通常のコード生成操作（GETINREG など）の全てに対する基本要素を持ち，さらに通常の分岐や条件文を持つ必要がある．この言語の設計はコード生成ルーチンや共通手続きの大部分が17.3節で述べた QD(i) や INS1, INS2 のような大域的変数を使用しているという事実によって可能となる．

この考え方を説明するために，410ページにある ＋ 用4-組に対するプログラ

ム片がどのようになるか，文字形式と内部形式で以下に示そう．各演算操作はそれを意味する整数といくつかのパラメタから成っている．操作はINIT初期設定(1)，FIXADルーチンの呼出し(2)，acc上にOPERANDをのせるコードの生成(3)，命令の生成(5)などである．機械語では50～75語となるプログラム片を20語に納めることができることに注目しよう．一つの操作を1語に納めることもあるいは可能かもしれない．

もしコンパイラがアセンブリ言語か良いマクロ機能をもった言語で書かれていれば，プログラム片の内部形式を貯える表は，文字形式の各命令をマクロ呼出しとすることによって文字形式から自動的に生成することができよう．

＋4-組に対するコード生成手続き

文字形式	内部形式	意　　味
INIT	1	INS1:＝0;INS2:＝0.
FIXAD 1	2,1	第1被演算子に番地を与えるためにFIXADルーチンを呼出す．
FIXAD 2	2,2	第2被演算子に番地を与えるためにFIXADルーチンを呼出す．
GETINACC 2	3,2	いずれか一方の被演算子をaccにのせるためにGETINACCルーチンを呼出す．
CHECKREG 2	4,2	第2被演算子の番地がINS2にあることを確かめる．
GEN 'ADD' 2	5,1,2	命令の生成．この操作は，命令の表の第1要素となる．番地部はINS2にある．
CHECKTEMP 1	6,1	2個の被演算子を検査する．中間結果であってその活きている範囲が終れば，その処理をする．
CHECKTEMP 2	6,2	
ENDSEG	6	終り．

より一般的なサブルーチンの作成

算術演算子や論理演算子に対する一般的なコード生成ルーチンを作る計画を示そう：(1)　被演算子の番地を作るコード生成，(2)　必要となる任意の型変換を行うコードの生成，(3)　被演算子の一つをacc上にのせるコードの生成，(4)　その演算子に対するコードの生成．

全ての算術演算子や論理演算子を取扱う一般的なコード生成手続きを作成することは可能なことが多い．この手続きの引数には，被演算子の記述，結果が有すべき型，演算が可換か非可換か，逆演算（逆除算など）が存在するか，個々の計算機特有の機能によって必要となるものなど，がある．重要な引数の一つは，骨組となる命令の系列で，これは上の(4)番目のステップに対する命令を生成する手続きによって使用される．

同じような考え方が IBM FORTRAN H コンパイラで用いられている（もっとも，こちらのコード生成機構は多少異なっている）．これをここで述べよう．

ビットの帯

IBM FORTRAN H コンパイラ（Lowry と Medlock (69) 参照）はコード生成法に関する情報を扱う表を小さくするために技巧をこらしている．まずレジスタは全て（IBM 360 と同じように）演算に用いることができると仮定しよう．

すでに前のパスで，非常に複雑なレジスタの大域的な最適化を行うような4-組が生成されている．各4-組（例えば，(+I T1, T2, T3)）に対して，どの被演算子がレジスタにあり，どの被演算子の値をそのレジスタに残しておかなければならないかなどがわかっている．$i=1, 2, 3$ として，既知のレジスタ R*i* は，必要とあれば，T*i* の値を保持するために，割付けられており，レジスタ B*i* は，T*i* の属するデータ領域の番地を保持している．（活きているデータ領域中にあれば，明らかに B*i* = 1.）このような情報は，被演算子とともに4-組の中に納められている．レジスタのあるものも同様であろう．例えば，もし T1 をレジスタの中に残しておく必要がなければ R3 = R1 としてよい．

さらに，4-組とともに以下のような情報を意味する8ビットの STATUS 欄が存在する（ビットが1であれば，対応する情報は真であり，0であれば，逆が成立つ）．

ビットの意味

1. 被演算子 T1 がすでにレジスタ R1 にある．
2. 被演算子 T1 の値はレジスタ R1 に残さねばならない．
3. 被演算子 T2 の値がすでにレジスタ R2 にある．
4. 被演算子 T2 の値はレジスタ R2 に残さねばならない．

5. T1 が属するデータ領域の番地はまだレジスタ B1 の上にない．
6. T2 が属するデータ領域の番地はまだレジスタ B2 の上にない．
7. T3 が属するデータ領域の番地はまだレジスタ B3 の上にない．
8. T3 は演算の後主記憶に退避しなければならない．

個々の演算に対して生成される命令は適当なビット系列を伴って，生成された順序に，骨組の形式でテーブルにおかれる．+I 演算子の例を図17.8に示す．コード生成の過程は，つぎのように働く：一つの一般的ルーチンにこの演算子に対する4-組と（それに関する全ての情報とともに）命令表を与える．このルーチンはつぎのステップを踏む．

1. 状態ビットの欄を指定するために8ビットの STATUS 欄の最初の4ビットを使用する——0000 が第1欄，0001 が第2欄，･･･を指定する．
2. STATUS 欄の後ろの4ビットと選ばれた欄の X とを置換える．最後の X がビット8，最後から2番目の X がビット7，･･･と置換わる．
3. ビットの欄を上から下にたどって，ビットが1ならば，それに対応する命令を生成する．

この手続きの働きを見るために，8ビットの STATUS 欄が 00000011 の場合を考えてみよう．これは T1 と T2 がともにレジスタ上になく，被演算子の値はレジスタ上に残しておく必要もなく，レジスタ B1 と B2 にはすでに番地がのっているが，B3 にはまだで，演算結果は退避すべきことを示している．最初の4ビットから第1欄が指定される．これは X010X010XX だから X を後ろの4ビットと置換えると 0010001011 となる．したがって，3番目，7番目，9番目，10番目の命令を順に生成すると

```
LREG   R3,D(B1)   R3 に T1 をのせる．
ADD    R3,D(B2)   T2 を R3 に加える．
LREG   B3,D(1)    T3 に番地を与える．
STORE  R3,D(B3)   結果を退避する．
```

オフセット D は被演算子の記述から決定される．またレジスタ番号は4-組の中に与えられている．

生成されたコードは，演算の可換性その他の特徴を考慮に入れて，骨組となる命令にさらに別の命令を追加すれば，もっと能率のよいものにすることができる．

骨組となる命令形式	状態ビット	
	0000000011111111	
	0000111100001111	
	0011001100110011	
	0101010101010101	
1 LREG B1, D(1)	xxxxxxxx00000000	T1を決める
2 LREG R1, D(B1)	0000111100000000	
3 LREG R3, D(B1)	1111000000000000	
4 MREG R3, R1	0000111100001111	
5 LREG B2, D(1)	xxxxxxxxxx00xx00	T2を決めて加算を行う
6 LREG R2, D(B2)	0100010001000100	
7 ADD R3, D(B2)	1000101010001000	
8 ADDR R3, R2	0111010101110111	
9 LREG B3, D(1)	xxxxxxxxxxxxxxxx	必要があれば結果を退避する
10 STORE R3, D(B3)	xxxxxxxxxxxxxxxx	

MREG はレジスタからレジスタへの移し換え.
ADDR はレジスタ上の値を他のレジスタに加える.

図 17.8 +I に対するビットの帯

17.6 目的モジュール

オブジェクト・デックあるいはバイナリ・デックとはカード上の目的プログラム（あるいは2次記憶上のカード形式の情報）である．主記憶上ならどこにでもこのプログラムをロードすることができれば，このモジュールは“再配置可能である”という．もしこれが，きめられた場所に必らずロードしなければならないのならば，“絶対形式である”という．この目的プログラムをロードする前に，他に必要な目的モジュール（入出力サブルーチン，標準関数，その他利用者が別々にコンパイルしておいたサブルーチンなど）と自動的に結合されるのが普通である．

再配置可能形式では，そのプログラム内の記憶場所を参照する命令や値は何らかの意味で，“再配置可能”でありその結果このプログラムが主記憶のどこにロードされても記憶場所の参照はその場所に応じて変更することが可能である．IBM 360 のような機械ではこの再配置可能を容易にするために，命令の中で記憶場所を参照するのに，基底レジスタを使わねばならない．ある種の“タイムシェアリング”計算機では，この再配置機構を完全にハードウェアで

実現しその結果，絶対モジュールと再配置可能モジュールの間に差がなくなっている．

再配置可能な目的モジュールの形式は，文字形式のアセンブリ言語の書式のように使用上融通性があるが，アセンブリ言語のようにアセンブルに要する余分な時間をもはや必要としない．しかしながら，目的モジュールを結合し再配置を行うこと自体時間がかかる仕事であることは指摘しておかなければならない．実行時間は非常に短く，大部分の時間がデバッグに費やされるような使用環境では，絶対形式のコードを生成して直ちに実行に入る方が適している．

IBM OS／360 の術語を使えば，いくつかの目的モジュールを結合して**ロード・モジュール** (load module) を形成する仕事は**リンケジ・エディタ** (linkage editor) によって行われる．ロード・モジュールを最終的に（再配置を伴って）ロードする仕事は**ローダ** (loader) によってなされる．（実際に，ロード・モジュール自身も退避したり後で他の目的モジュールやロード・モジュールと結合したりできる．）もっと簡単なオペレーティング・システムのもとでは，上の2段階のステップは**結合ローダ**とか，単にローダと呼ばれる単一のプログラムで処理されている．

ここでは IBM OS／360 の目的モジュールの基本形式を述べる．ここで述べる以外の詳細は，IBM(d) や Ehrman を参照されたい．他の計算機におけるソフトウェア体系も似たような――多少の差こそあれ融通性のある――形式を用いていよう．しかし術語の定義は大幅に異なっている可能性がある．

単純で，理解しやすい書き方にするため，目的モジュールの形式を幾分制限して細かい点のいくつかは省略する．この節のカードの図で明示していない欄は空白である．73～80欄は識別と番号付けのために使用できる．数字は読み易さを考えて10進法を用いる．実際には16進文字を使用すべきものである．

主たる関心はつぎのような点にある：

1. この目的モジュールが必要とするのはどの外部サブルーチンかをどのようにして明示するか．
2. “エントリ・ポイント”（他のサブプログラムが参照できるこのモジュールに含まれる命令やデータの場所）をどのように明示するか．
3. どのようにして目的プログラムを再配置可能に作り主記憶上のどこにでも置けるようにすることができるのか．
4. 目的モジュールの形式を利用してコンパイルを容易にしたり高速にした

りすることが可能か.

CSECT

目的モジュールは主として**テキスト**から構成されている. これは目的プログラムを形成している機械語の命令とデータの集りである. モジュールは, テキストの部分セクションを1個以上含むことができる. これは**コントロール・セクション** (control section) あるいは**CSECT**と呼ばれる. CSECT の内部では, テキストの各バイト位置 (CSECT の先頭からの相対位置) が固定されていて, この順序はCSECTが主記憶上にロードされるときもそのまま維持されねばならない. しかしながら, 個々の CSECT の順番は, 適宜でよく, CSECT はそれぞれ主記憶のどこに置かれてもよい.

CSECT の通常の用法は

1. 主プログラムに対する命令の系列
2. サブルーチンに対する命令の系列
3. 主プログラムあるいはサブルーチンに対する各データ領域
4. プログラムの定数群

などである. しかしながら, このような分類にこだわる必要はなく, テキストのどんな部分セクションも, それが便利ならば一つの CSECT に置いて構わない.

目的モジュール・カードの型

目的モジュールは四つの部分から成り, それぞれ図17.9に示すような順序でなければならない. それぞれの部分はつぎのとおりである.

1. ESD (external symbol dictionary, 外部記号辞書). これは下記の項目を定義し識別する (一意的な名前と番号を与える):
   a) 目的プログラム CSECT (複数個ありうる).
   b) CSECT への外部参照とこのモジュールで使用されたり呼ばれたりしているがそれ自身の中には含まれていない, よそのエントリ・ポイント. これらはリンケジ・エディタによってこのモジュールと結合されるもの.
   c) エントリ・ポイント——他のモジュールから参照され得るこのモジュール内部の場所. 他の目的モジュール中での外部参照としてエント

リ・ポイントの名前が現れると**この**モジュールがそのモジュールに結合される．OS／360 では，これを**名札定義** (label definitions) と称している．

d）無名および名前付 COMMON 領域（主として FORTRAN で使用）．

2. "TXT" カード上のテキスト，これは目的プログラムの実際の機械語命令とデータを与える．
3. RLD (relocation dictionary, 再配置辞書)．これはテキスト中に現れる再配置すべき番地を記述する．
4. END カード．目的モジュールの終りを示す．



図 17.9　目的モジュールの形式

外部記号辞書 (ESD)

各 ESD カードには，1個の ESD 項目——CSECT, 外部参照，エントリ・ポイント，COMMON 領域——が記述されている（図 17.10 参照）．17—24欄にはプログラマが明記したルーチンまたは領域の文字通りの名前（以下英字名と称する）が記入されている（さらにシステムによって内部的に必要なルーチン——入出力ルーチン, 標準関数など——の英字名も記入される）．無名 COMMON の場合は空白が入る．

エントリ・ポイントを除く各項目はまた **ESDID** (ESD 名) と称される16ビットからなる一意的な ESD 番号を持つ．ESDID は，このモジュールを実際に組立てた人によって割付けられ，テキストや RLD が内部的に項目を参照する時に使用される．英字名を使うより場所が節約できる．同一モジュール内でエントリ・ポイントを参照するのに英字名を使うことはありえないので，エントリ・ポイントには ESDID を与えない．

各項目は，図 17.10 に示すように第9欄を境に分割されている．

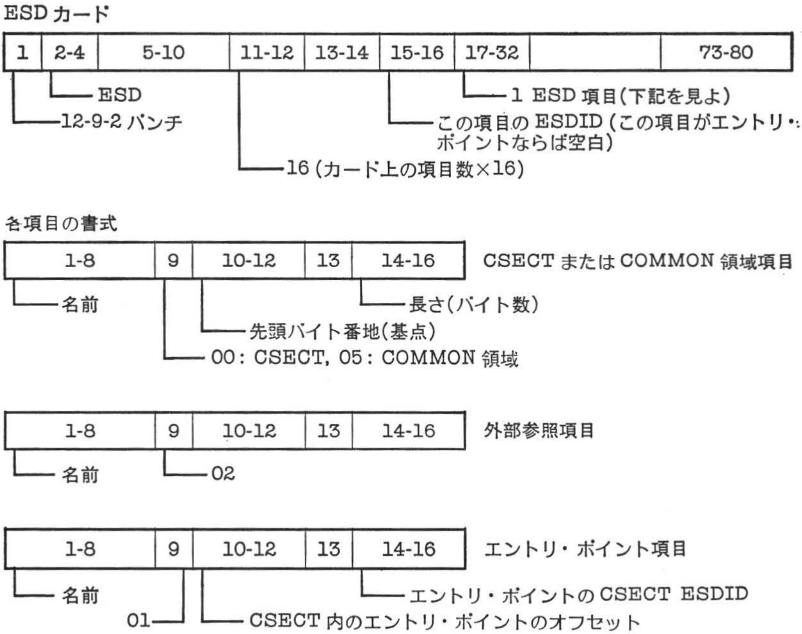


図 17.10 ESD（外部記号辞書）カードの書式

ほかにどんな情報が必要か．CSECT や COMMON 領域にはその長さ――割付けに必要なバイト数――を指定しなければならない．これはその項目の14―16欄に記入する．さらに**基点**（origin）と呼ばれる，領域の先頭番地を示す情報も10―12欄に記入する．領域内における テキストの 全ての 番地（オフセット）はこの基点との関連で与えられる．したがって，基点が 500 であれば，オフセット 500 は領域の先頭バイト，501 は次のバイトなどとなる．基点は 0 となることもある．

外部参照の場合は，この名前によって参照されたCSECT はこのモジュールの外にあり，名前のほかは未知なのでほかの情報を与えることはできない．

エントリ・ポイントに対しては 2 通りの情報が必要である．一つはそれがどの CSECT に置かれているかを知らねばならぬことで，これはこの項目の14―16欄に CSECT の ESDID を置くことで示す．もう一つは CSECT 内でのエントリ・ポイントのオフセット（基点に関して）を知らねばならないことであ

る．これは10—12欄に示す．

図17.11は ESD の1例で，これは A という名前の CSECT（長さ100，基点0），B という名前の CSECT（長さ200，基点501），SIN と COS ルーチンの外部参照，無名COMMON（長さ100，基点0），C1 という名前付 COMMON（長さ300，基点0），CSECT A 内第20バイト目のエントリ・ポイントA1，CSECT B 内第101バイト目のエントリ・ポイント B1 をそれぞれ記述している．各エントリ・ポイントはそれぞれの CSECT の基点に関連して与えられていることに注意．

1~4	11~12	15~16	17~24	25	26~28	30~32
$ESD	16	01	A	0	0	100
$ESD	16	02	B	0	501	200
$ESD	16	03	SIN	2		
$ESD	16	04	COS	2		
$ESD	16	05		5	0	100
$ESD	16	06	C1	5	0	300
$ESD	16	07	A1	1	019	001
$ESD	16	08	B1	1	601	002

図 17.11　ESD の例

テキスト（TXT）カード

テキスト（TXT）カードの書式はつぎのとおり：

1	2~4	6~8	11~12	13~14	15~16	17~72	73~80

- 1：12-9-2 パンチ
- 2~4：TXT
- 6~8：テキストの先頭バイトの番地（CSECT 内）
- 11~12：カード上のテキストの必要バイト数
- 15~16：テキストの属する CSECT の ESDID
- 17~72：テキスト

各カードには，そのカード上のテキストが属している CSECT または COMMON 領域の ESDID と，そのテキストの CSECT 内における番地（その ESD 内のその項目で指定される基点から相対的なもの）が含まれている．テキストの先頭バイトは，ここで指定したバイトに置かれ，以下引きつづくバイトに置かれる．CSECT の全部または一部に対するテキストがない場合は，そこに対応するバイトは，実行が開始される時点では未定義である．

リンケジ・エディタは，CSECT 内のテキストカードがオフセットの大きくなる順に並んでいれば能率よく働くことができる．このように並べることは要請されてはいないし，時には，必ずしもこの順に従わずにコンパイル時にテキストを出力することができることがある．また同一番地に対するテキストが複数枚のカードに現れることもできる．この場合は，そのようなテキストの**最後の**ものがその以前に現れたテキストを全て無効にしてしまう．図 17.12 にこれを示す．(大きなプログラムを修正して，アセンブルし直すのは時間がもったいないので，デックに“つぎを当てる”ためにしばしばプログラマが用いるもの．プログラマが誤りを1個見つけると，目的モジュールの書式でカードに命令をパンチする．このようなカードをまとめて目的モジュールデックの後ろに置き，対応する内容を書変えプログラムの誤りのある部分を変更する．†)

	番地	所要バイト数	ESDID	テキスト	
0~4	6~8	11~12	15~16	17~72	
$TXT	000	56	1	A……A	(56 個の A)
$TXT	056	56	1	A……A	(56 個の A)
$TXT	112	56	1	A……A	(56 個の A)
$TXT	001	2	1	BB	
$TXT	010	10	1	CCCCCCCCCC	
$TXT	000	2	1	DD	
$TXT	167	1	1	E	

上のカードは下記のような結果をもたらす

```
000 DDBAAAAAAACCCCCCCCCCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
056 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
112 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAE
```

図 17.12 テキストカードの例

コンパイラは一定の大きさの記憶場所を設けて，これに生成中の目的プログラムをのせる．この“バッファ”がいっぱいになると，その内容を2次記憶に書出さねばならない．しかしその中にある情報が全て既知になっているとは限らない．例えば，前進方向の分岐などがそうである．これを解決するにはコード生成時の最後に目的プログラムを読込んで，このような情報を埋めるのが普

† **訳注** ここではこのような修正が可能であることを述べており，現在この方法がしばしば用いられているというわけではない．

通である．

このようにすると 2 次記憶への読み書きの回数が増えるので，この分の仕事をリンケジ・エディタに行わせて節約を図ることもできる．これはコンパイラよりずっと小さいので，目的プログラム全体が主記憶の中に入ってしまう場合もある．

目的プログラム用のバッファがいっぱいになると，TXT カードの書式で書出す．未知だった情報が手に入るとこの情報を伴った TXT カードの書式を書出して，先に置かれていた間違ったテキストを無効にする．この後始末もリンケジ・エディタには多少面倒なことかもしれないが，これを用いればコンパイル時間を節約できる．

各テキストカードには，その テキストが 属する ESDID が 含まれているので，別の CSECT に対するテキストカードが混入してもよい．これを使うと，特に1-パスコンパイラでは便利である．すなわち主プログラム，各手続き，およびこのプログラム中で使用する諸定数に対するコードを，プログラムを一度構文解析しただけで生成するものだからである．

再配置辞書 (RLD)

目的モジュールの中では，テキストのバイト番地は

(ESDID，オフセット)

の形で明記される．ESDID は，この番地が入る CSECT または COMMON 領域の番号で，オフセットは，この ESD 項目の中で定義されたものとして基点に連らなる領域内の番地である．このような番号の対は，すでにエントリ・ポイントの定義で使用していた．プログラムをロードする最中に，CSECT あるいは COMMON 領域は主記憶上に場所が割当てられる．ローダはこの番地の対をつぎのような絶対番地に変換する：

CSECT あるいは COMMON の機械番地
＋(オフセット－ESD 中で定義されている基点)

例えば，CSECT 2 の第20―23バイトには番地の対 (2，100) ――CSECT 2 のバイト番地100――が入るべきものとしよう．さらに CSECT 2 の基点を10とし，CSECT は1000番地からロードされるとしよう．すると第20―23バイトに置かれる機械番地は

$$1000 + (100 - 10) = 1090$$

となる．今述べた再配置型番地と第8章で述べたデータ領域中の変数をコンパイル時番地（データ領域，オフセット）で表したこととの間の類似点に注意してほしい．

RLD（と場合によってはテキスト自身）は再配置されるべき番地を全て記述するために使用される．RLD 項目にとってどんな情報が必要か．知らなければならないのは再配置番地（参照されるバイトの番地）と再配置された番地を置くべき場所（テキスト中の**位置**）である．これも（ESDID，オフセット）の形式で与えられる．もう一つ知らなければならないのは，再配置された番地に必要なバイト数（**位置の長さ**）である．つまり下記の5通りの数値が必要である．

1. 再配置　ESDID ⎫ 再配置可能番地
2. 再配置　オフセット ⎭
3. 位　置　ESDID ⎫
4. 位　置　オフセット ⎬ 再配置された番地が入る位置
5. 位置の長さ ⎭

例えば

1. 再配置　ESDID　　＝2
2. 再配置　オフセット＝20
3. 位　置　ESDID　　＝3
4. 位　置　オフセット＝50
5. 位置の長さ　　　　＝3

としよう．二つの ESD 基点をともに0とすると，この数値は CSECT 3 の50，51，52 の3バイトに，CSECT 2 内のバイト 20 の機械番地が入るべきことを明示している．

図17.13に RLD のカード上の書式を示す．1枚のカード当り，7個までの番地＝RLD 項目を示すことができる．実際，2種の項目があり，一つは内部 RLD 項目で，位置が，モジュール内で定義された CSECT や COMMON 領域を参照している場合に用いられる．この場合再配置 ESDID と，位置 ESDID，位置オフセット，位置の長さとは同じ RLD 項目の中で定義される．再配置オフセットはどこにあるか．場所を節約するために，<u>これは RLD 項目の中で</u>

定義された位置にあるテキスト（TXT）カードの中に書かれている．

もう一つの RLD 項目は，再配置 ESDID が外部参照の場合に用いられる．位置の長さは 4 バイト（＝1語）でなければならない．また再配置オフセットは自動的に 0 である．この第二の型は，オーバレイの問題が生じた時に必要となる．これについてはここでは述べない．

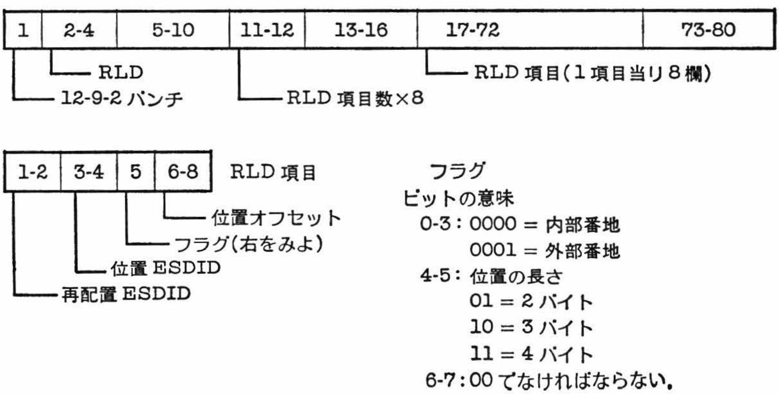


図 17.13　RLD カードの書式

ま と め

以上の議論で触れずにおいた点がいくつかある．1 枚の ESD カードには ESDID 項目が 3 個まで現れてよいこと，各 RLD 項目に必要な場所を減らす方法があること，各モジュール内の一つの CSECT の長さを示すのに ESD 項目の中ではなく END カードでよいこと．これは便利であるが，テキストカードの順番に対する制限をつけることになる．

オーバレイの問題も議論しなかった．これは数個の副プログラムからなるプログラムが必要に応じて主記憶上で入れ代って交代して実行されるものである．また，別個のモジュールに同一の CSECT あるいは COMMON 領域が含まれている場合のリンケジ・エディタはどんな処置をとるか．

OS／360 の目的モジュールでは ESD 項目にさらに二つの型を許している．これは“個人用コード”(PC) と“疑似レジスタ”である．後者は PL/I コンパイラで十二分に使用されている．これについては IBM(d) や Ehrman を参照されたい．

演習問題　17.6

1.　この節で述べたような目的モジュールを結合するリンケジ・エディタを設計し作成せよ．その前に COMMON 領域や，CSECT やエントリ・ポイントが二重に定義された場合の処置を決めておくこと．

2.　ローダを設計し作成せよ．リンケジ・エディタの出力は目的モジュールの形式と同じと仮定せよ．ただし，全ての結合はリンケジ・エディタにより終っているので，外部参照は存在しないものとする．

第18章

目的コードの最適化

目的コードの最適化とは，コンパイル中のプログラムの演算の順序を変えたり，内容を変えたりして目的プログラムをより能率のよいものにする処理過程をいう．最も優れた最適化コンパイラは FORTRAN や ALGOL で書いた込入ったプログラムに対する目的プログラムを生成するものであるがこれは熟練者が同じものをアセンブリ言語で書いたと同程度に能率よく働き，しかも人間よりずっと費用をかけないで実現することができる．

われわれは最適化技法を2通りに分類する：一つは原始プログラム（内部形式プログラム）に施すもので目的言語には依存しない．もう一つは目的プログラムに対して施すものである．特に後者は目的言語に強く依存しており，ここで余り詳しく論ずるつもりはない．18.4 節で手短かに議論し参考文献を示している．

第一の類に属する技法は，FORTRAN や ALGOL や PL/I などほとんどの算法言語に適用できるものである．主な方法はつぎの4通りである．

1. 畳み込み：コンパイル時に既知の被演算子を含む演算を実行すること．
2. 冗長な演算の除去（大抵の場合，適当な共通式を括り出す）．
3. ループの中で値の変化しない被演算子を持つ演算をループの外に移動．
4. ループ中に含まれる乗算の強さを減ずる（結局は乗算を加算に変換）．

コンパイラが施す最適化の程度は多くの原因に左右される．どんなコンパイラでも，できるかぎり完全な最適化を行いさえすればよいというものではない．例えば，プログラムの検査が主な用途であるようなコンパイラでは余りに高度な技巧をこらすべきではない．何となれば，それによって演算の順序が根本的に変ってしまいデバッグをかえって難しくしてしまう．

したがって，ここでは完全な最適化のための広汎な計画を説明するつもりは

ない．その代りに，18.1 節では二つの簡単な技法から始めよう．その一つはどんなコンパイラでも採り入れるべきものである．18.2 節では for ループの最適化を行う中程度に複雑な方法を議論する．とはいってもかなり能率よく実現できる．18.3 節では，原始プログラムの構造の解析を一層完璧に行う必要のあるもっと複雑な最適化に取組む．この節では実際に計算機で実現する方法についてはあまり詳細には渡らない．この一層複雑な方法を自分で作成してみようという読者は，とにかくこの仕組に関する文献をよく研究してほしい．18.4 節は文献を概観し，前の 3 節で触れなかったこのほかの技法を簡単に紹介する．これには目的言語に依存する方法もある．

本章を通じて最適化技法の説明には 3-組か 4-組のいずれかを用いる．どちらを用いるかはその時の便利さによる．どちらの内部形式が良いかを言うことは難かしく，まだだれもこの問題を実際には研究していない．あるのは教えられた意見ばかりである．一般にいって，著者自身は，3-組あるいは間接 3-組が好きである．何となれば 4-組ではいつも中間結果の値を記述しなければならないからである．しかし，3-組の方が，これから見るように，処理上問題がたくさん生ずる．読み易さを考えて 3-組や 4-組の代りに通常の ALGOL によるプログラムを書くことが多いだろう．

18.1 基本ブロック内における最適化

11.5 節で定義したように（基本）ブロックとは，順番に実行される演算の系列で，入口と出口——それぞれ最初と最後の演算——をそれぞれただ 1 個持っている．つぎのような代入文

(18.1.1)　　`I := 1 + 1; I := 3; B := 6.2 + I;`

の系列が上に述べたブロックの例である．2 通りの最適化——畳み込みと冗長な演算の除去——は，通常，ブロックの内部で行われる．

畳み込み法

畳み込み (folding) とは，原始プログラム上で被演算子の値が既知の演算をコンパイル時に実行してしまう処理方法で，この演算は実行時に行う必要がなくなる．例えば **図 18.1 a** のプログラム片を考えてみよう．このプログラムの 3-組内部表現は **図 18.1 b** である．明らかに 3-組 1 はコンパイル時に実行でき結果の定数値を置き換えることができる．それほど明らかではないが，3-組

(a) プログラムの一部	(b) 3-組の系列	(c) 最適化された3-組の系列
I := 1+1;	(1) + 1, 1	(1) := 2, I
I := 3;	(2) := (1), I	(2) := 3, I
B := 6.2+I;	(3) := 3, I	(3) := 9.2, B
	(4) CVIR I	
	(5) + 6.2, (4)	
	(6) := (5), B	

図18.1　畳み込みの例

4も実行できる．それは I の値が3と定まっているからである．したがって3-組5も実行できることになる．畳み込み法を施した最終結果が**図18.1c**である．

畳み込みは主に4則演算子 +，−，*，/ に適用される．それらが原始プログラム中に現れるのが最も多いからである．また 図18.1 の CVIR（整数から実数への変換）のような変換演算子への適用も当然である．これは暗に示されているよりは図18.1 のように内部表現中に実際に変換が現れる方が簡単に行える．

被演算子が定数である演算の畳み込みの処理は理解しやすくこれ以上説明の要はない．被演算子の値が別のところで定まる演算の畳み込みは多少難かしい．これは普通は基本ブロックの内部だけで行われており，初期状態が空である表 T を使用する．畳み込みの処理中 T には既知の現在の値 K を持った全ての単純変数 A が順序対 (A, K) として入っている．さらに，畳み込み処理で，畳み込みを受けた3-組を新しい3-組 (C, K, 0) と置き換える．C（定数を表す）は新しい演算子で，この3-組からはコードは何も生成されない．K は畳み込みの結果得られた定数値である．畳み込みのアルゴリズムは，基本ブロック中の3-組を先頭に処理する．各3-組の処理の模様はつぎのとおり：

1. 被演算子が T 表中の変数であれば，それを対応する値 K と置換える．
2. 被演算子が (C, K, 0) という形式の組を参照していれば，それを定数 K と置換える．
3. 全ての被演算子が定数で演算が畳み込み可能ならばその3-組の演算を実行しそれを3-組 (C, K, 0) と置換える．ただし K は実行の結果を

示す．

4. 3-組が代入文 A := B で A が単純変数であれば
   a) B が定数の場合，T 表の A の値として B を書入れる（必要ならば A の古い値は表から削除する）
   b) B が定数でない場合，T 表に A があればその既知の値とともに A を削除する．

このアルゴリズムを 図 18.1 の基本ブロックに適用したものが下の **図 18.2** である．

サブルーチンや関数を呼出している 3-組 を 処 理 する時は注意が必要である．FORTRAN のコンパイルでは，T 表からその呼出しの全ての実引数と全ての COMMON 変数とを削除しなければならない．というのはその呼出しによってそれらの値が変更される可能があるからである．ALGOL では，もしその手続き呼出しの結果を完全に知っているのでなければ，T 表は完全に空にしなければならない．また入力文でも変数に値が与えられることに要注意．

さらに，畳み込みを実施したことによってコンパイル時に何らかの誤りが生成されないことを確認しなければならない．例えば，プログラマが 1/0 という式を書けば，コンパイラは分母が 0 であることを検出するに違いない．不幸なことに，このような演算があっても必ずしも，その目的プログラムが，間違って動くことを意味するわけではない（もっとも現実のプログラムではその場合が多いが）．例えば，原始プログラムが

```
IF FALSE THEN A := 1/0;
```

という実行文を含んでいても，実行時には誤りは発生しない．演算の畳み込み

1	C 2	C 2	C 2	C 2	C 2	C 2
2	:= (1), I	:= 2, I	:= 2, I	:= 2, I	:= 2, I	:= 2, I
3	:= 3, I	:= 3, I	:= 3, I	:= 3, I	:= 3, I	:= 3, I
4	CVIR I	CVIR I	CVIR I	C 3.0	C 3.0	C 3.0
5	+ 6.2, (4)	+ 6.2, (4)	+ 6.2, (4)	+ 6.2, (4)	C 9.2	C 9.2
6	:= (5), B	:= (5), B	:= (5), B	:= (5), B	:= (5), B	:= 9.2, B
T:		(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (B, 9.2)

図 18.2 畳み込みの段階的実施状況

がコンパイル時に誤りを発生するような場合は，警告の文句を印刷しその演算は畳み込みをしないままにしておくのが最善である．

3-組 となった内部形式を別のパスで畳み込む方法を見てきたが，間接 3-組や 4-組 を使用することもできた．このほか内部形式あるいは機械語を生成する意味付けルーチンの中で畳み込みを行う可能性もあり，ここでは，パスを分ける必要はない．例えば，上向き構文解析を仮定すると

$$E^1 ::= E^2 + T$$

に対するルーチンでは E^2 と T の意味の検査しか必要がない．両者がともに定数であるか，その値が既知となっていれば，このルーチンは，それらの値を加えて結果を E^1 と結び付ける．これらの値と変数を保持する表も使用することができる．しかしこの表は各名札を構文解析する個所（すなわちコードが生成されて分岐先となるような個所で）で空にされなければならない．

冗長な演算の除去

基本ブロック内で i 番目の演算が**冗長である** (redundant) とは，その前に（j 番目とする）同等な演算があって，j 番目と i 番目の間にある第三の演算が，i 番目の演算が依存している変数のいずれをも変更しないことをいう．例えばD := D + C * B; A := D + C * B; C := D + C * B; からなる基本ブロックを考えよう：

(18.1.2)	(1) * C, B	(4) * C, B	(7) * C, B
	(2) + D, (1)	(5) + D, (4)	(8) + D, (7)
	(3) := (2), D	(6) := (5), A	(9) := (8), C

2回目，3回目に現れる C * D は冗長である．つまり 3-組 (1) のあとでは C も B も値が変化していない．3-組 (5) の C * B に D を加える演算も C * B が不変であるから冗長であると考えるかもしれない．しかしこれは違う．何となれば最初の加算 (2) の後で，3-組 (3) で D の値が変更されているからである．3番目の D + C * B(8) は冗長で，3-組 (5) の参照で置換えることができる．

普通，プログラマは冗長なプログラムを書かないように教えられているので，冗長な演算を除去してもその効果は疑問かもしれない．しかし，冗長なプログラムを書くのは，容易だし，プログラムが読み易くなることが多い．さら

に，冗長な演算にはいろいろあって，多分，添字式に現れるようなものは，プログラマが制御しようのないものである．例えば代入文 X[i, j] := X[i, j + 1] を 3-組形式で考察してみよう．†

(1) ∗ i, $d2$	(3) ∗ i, $d2$	(5) + (4), 1
(2) + (1), j	(4) + (3), j	(6) := X[(5)], X[(2)]

明らかに 3-組 (3) と (4) は冗長で除去が可能であり，結局つぎのようになる：

(1) ∗ i, $d2$	(3) + (2), 1
(2) + (1), j	(4) := X[(3)], X[(2)]

冗長な演算を除去するアルゴリズムは並んでいる順に 3-組を走査する．3-組 j があるために 3-組 i が冗長となれば，i は (SAME, j, 0) という 3-組で置換わる．ここで SAME は何もしない演算子 (NO-OP) であり，これに対するコードは何も生成されない．変数と 3-組の依存関係を跡付けるために，つぎのようにして，これらに**依存番号** (dependency number) を割当てる：

1. 変数 A の依存番号 dep (A) の初期値は 0．何となればその初期値は，どの 3-組にも依存しないからである．
2. 3-組 i を処理した後に，それが A に値を代入しているならば，dep (A) を i に変える．即ちこの変数の値は 3-組 i に依っていることを示す．
3. 3-組 i を処理している際に，その依存番号 dep (i) は

   1 + (その被演算子の依存番号の最大値)

   で定義される．

依存番号の用法はつぎのとおり：3-組 i が 3-組 j (ただし $j < i$) と同等であれば dep (i) = dep (j) のときに限り 3-組 i は冗長である．

これはつぎのように示すことができる．定義から 3-組 j の被演算子の依存番号は j 未満である．3-組が依存している変数はいずれも j と i の間で値が変化しないとしよう．すると 3-組 i は冗長である．この場合 3-組 i の被演算子の依存番号は，3-組 j, ⋯, $i-1$ が処理されている間に変化すること

† 訳注　ここでは 2 次元の配列は行方向に記憶されているものとしている．

がないことに注意．したがって dep $(i)=$ dep (j) となっている．

ところが，i が依存している変数 A が 3-組 k $(j \le k < i\cdots)$ で変化するとしよう．この場合 dep (A) は k $(k \ge j)$ に変る．3-組の依存番号を決める方法から dep $(i) >$ dep (A) $= k \ge j \ge$ dep (j) であり，i と j の依存番号が異なる．

冗長な演算を除去するアルゴリズムは，つぎのようにして各 3-組を順番に処理する：

1. その 3-組の被演算子が，3-組 (SAME, j, 0) を参照していれば，その被演算子を (j) で置き換える．
2. 3-組 i の依存番号を計算する：dep(i) := 1 + (二つの被演算子の依存番号の最大値)．
3. 3-組 i と同じで dep(j) = dep(i) なる 3-組 j (ただし $j < i$) が存在すれば，3-組 i は冗長である．この 3-組 i を (SAME, j, 0) に変更する（この検査は，$i-1, i-2, \cdots, 1$ の順に比較することによって行うこと．ハッシュ技法を用いると有利である）．
4. 3-組 i が，配列 B の要素あるいは単純変数 B に値を代入するものならば，dep (B) に i を設定する．

図 18.3 は，基本ブロック (18.1.2) を用いて，上のアルゴリズムを段階的に適用していった結果を示したものである．各段では，もとの 3-組と，このステップを実行する直前の各変数の依存番号，それから計算したその 3-組の

処理すべき 3-組	変数の依存番号	3-組の依存番号	得られた 3-組
	A B C D		
(1) * C, B	0 0 0 0	1	(1) * C, B
(2) + D, (1)	0 0 0 0	2	(2) + D, (1)
(3) := (2), D	0 0 0 0	3	(3) := (2), D
(4) * C, B	0 0 0 3	1	(4) SAME 1
(5) + D, (4)	0 0 0 3	4	(5) + D, (1)
(6) := (5), A	0 0 0 3	5	(6) := (5), A
(7) * C, B	6 0 0 3	1	(7) SAME 1
(8) + D, (7)	6 0 0 3	4	(8) SAME 5
(9) := (8), C	6 0 0 3	5	(9) := (5), C
	6 0 9 3		

図 18.3　冗長な演算の除去

依存番号，その段で得られた 3-組を示している．

被演算子が添字付変数 B[T] の場合は，その 3-組の依存番号は B と T の両方に依存する． さらに，3-組 i で添字付変数 B[T] に代入が行われていると，dep(B) を i に変更する．したがって，以後に B を参照するときはつねにこの 3-組に依存する．

間接 3-組 を普通の 3-組 に代って使用 すると 冗長な演算の除去が簡単になる．つまり 3-組 が同じものはすでに検知済みだからである．例えば， 基本ブロック (18.1.2) を間接 3-組で表してみよう：

1.	(1)	6.	(4)		(1)	*	C, D
2.	(2)	7.	(1)		(2)	+	D, (1)
3.	(3)	8.	(2)		(3)	:=	(2), D
4.	(1)	9.	(5)		(4)	:=	(2), A
5.	(2)				(5)	:=	(2), C

OPER 表　　　　　　3-組 表

ここでは， OPER 表の演算子の代りに， 依存番号と 3-組 とを結び付ける． 基本ブロックを処理する 前 に 各 3-組 の 依存番号を 0 にしておき，dep (i) は 3-組 i を参照する OPER 表の演算が処理されるたびに変更される．

畳み込みと冗長演算の除去に関する議論

ここで与えたアルゴリズムは最善のものではない．演算が可換なものに対してはそれを利用すれば，アルゴリズムをもっと良いものにすることができる．例えば，A*B と B*A は同一のものと見なしてよい． これを組織的に行うために， n 項＋(あるいは n 項*) の被演算子を全て規則的に並べなければならない．すなわち，まず，変数でも定数でもない項を並べ，つぎに添字付変数を ABC 順に並べ，そして単純変数を同様に並べて，最後に定数を並べる．第 11 章で述べた内部形式はいずれも，これを含むような拡張が可能である．例として，

A := 1 + B + C + 2;　B := B + C + 6

を並べ換えて

A := B + C + 1 + 2;　B := B + C + 6

とすると 1＋2 が畳み込めて 2 番目の B＋C が除去できるようになる．しかし，これで万事うまくゆくわけではない．

A：＝ B＋C＋B＋C

を並べ換えて

A：＝ B＋B＋C＋C

とすると B＋C を一度だけ計算すればよいことが検出できなくなる．共通的な部分を全て発見することは式中の部分式の括り方が何通りもあるので実際問題として大変難かしい．この問題については 18.4 節でさらに議論する．

単項演算子 ABS と－を外に追い出すのは 17.4 節で説明したようにコード生成時ではなくて，この時点でも行うことができる．これによって単項演算を省略できるばかりでなく，さらに冗長な演算を発見できることもある．例えば，単項 － を ′ で表わして実行文 C：＝ A－B; D：＝ B－A; を 3-組 で書くと

(1)　＋　A, B′	でありしたがって	(1)　＋　A, B′	
(2)　：＝　(1), C		(2)　：＝　(1), C	
(3)　＋　B, A′		(3)　＋　A, B′	
(4)　：＝　(3), D		(4)　：＝　(3)′, D	

と変換される．これによって同じ演算を発見できる．被演算子の符号を逆転する機能を表す欄を設ける方法が，冗長な演算の除去を行わない場合でも，よく用いられている．何となれば，ある機械では，“store negative”（符号を逆転して主記憶にしまう）とか “load negative”（符号を逆転して acc にロードする）という命令がありこれを用いて目的コードを短くすることができるからである．

もとの 3-組 形式では，各 3-組 は，高々一度しか参照されない．同様に，4-組 を使うと，コンパイラが生成した中間結果用変数 T は被演算子として一度参照されるだけである．これは通常の代入文を内部形式に翻訳するときは自然に生ずる．したがって中間結果の活きている範囲を決定するのはやさしい．しかしながら，このような最適化処理を行った後では，ある 3-組が，何度も参照される可能性がでてくる．こうなると，その 3-組の結果が活きている範囲を決定するのが一層難かしくなる．詳細は 14.2 節を参照されたい．

18.2 ループの中程度の最適化

for ループ中の演算が**不変である** (invariant) とは，その演算が依存している被演算子がいずれも，ループの実行中に，変化しないことをいう．効果的な最適化を行うと，不変な演算をループの外に移動できる．例えば，実行時に 1000 回繰返されるループ中の 1 個の乗算をコンパイル時にループの外に出しておけば 999 回の実行時の乗算を節約できる．

ここで述べるループの最適化の第二の方法は，**演算の強さを減ずる** (reducing the strength of an operation) と称されるもので，これは，この演算をより速く実行できる演算と置換えるような意味である．主たる関心は I がループの制御変数の場合に I * K のような乗算を加算に変更するところにある．これを示すために，つぎのループを考えてみよう：

```
FOR I := A STEP B UNTIL C DO
    BEGIN ...    T1 := I * K ...    END
```

ここで K はループ内で不変であるとする．I は初期値 A を代入されて，ループ内で，**再帰的に定義**される (recursively defined)．再帰的定義とは，右辺に自分自身を含むからである (すなわち，ループ中での I への代入は，ただ1ヶ所で I := I + B である)．I が値を B だけ変化するたびに，I * K (と T1) の値は B * K だけ変化する．すなわち

$$(I + B) * K = I * K + B * K$$

である．したがって，もし T1 がループ中ここ以外では変更を受けないならば，I * K の演算の強さを減らしてループをつぎのように変更する．

1. ループの直前に T1 := A * B; T2 := B * K を挿入する．ただし T2 は新しい中間結果用変数．これによって T1 を初期設定し B * K を予め計算しておく．
2. ループ中から T1 := I * K を削除する．
3. ループの最後の文として T1 := T1 + T2 を挿入する．

かくてつぎのプログラム片が得られる：

```
T1 := A * K;  T2 := B * K;
FOR  I := A STEP B UNTLI C DO
```

```
BEGIN···    ···T1 := T1 + T2; END
```

ループ中の乗算を効果的に加算に置換えることができた．もちろん，ここで A と B はループ中で不変，I はループ中で，通常の増分以外の変更はないものと仮定している（実際問題として，この方法では，演算の順序を多少変更しており，A がループ中で不変かどうかは問題とならないだろう）．

注意を一言．B と K は整数型と仮定していた．K が実数型ならば，乗算の強さを減じてはならない．何となれば，置換わった加算は，浮動小数点数を用いているので丸め誤差のために精確ではないからである．この丸め誤差は，ループを繰返すにつれて，打消し合うよりは累積する傾向が強い．一般に，K + K + ··· + K（I 回）よりは I*K の方がずっと精度が高い．

乗算の強さを減ずることは乗算が加算と同程度に速い場合でも有用である．何故ならばこれによって，同じ演算が何度も現れるのを発見してこれの除去が容易にできるようになるからである．したがってループの内では限定された形式の冗長な演算の除去を行っている．これは添字付変数の評価を最適化する場合に最も有用である．

乗算の強さを減じても必ずしも最適化されないことがあることは注意しておく必要がある．実際，プログラムの実行速度を遅くすることさえある．次ページの左側のプログラムを考えてみよう．ループ内の実行文を実行するたびに，乗算が 1 度行われる．演算強さの軽減（と畳み込み）によってプログラムを右側のように変えると乗算はループ中で一度も実行されず加算が 10 回行われることになる．

演算強さの軽減は大抵の場合プログラムの能率を向上させる．しかし，ループの中がいくつかの部分に分れていて，ループを通るたびに，それらのうちの一部しか実行されないような場合は，能率がよくならない可能性がある．例えば，図 16.1 のようなインタプリタを最適化すると（ここではループはかげにかくれている）このインタプリタの能率は悪くなるおそれがある．

```
FOR I:= 1 STEP 1 UNTIL 10 DO
   CASE I OF
      K:= I * 1 ;
      K:= I * 2 ;
       :
      K:= I * 10 ;
   ENDCASE;
```

```
T1:= 1; T2:= 2; … T10:= 10;
FOR I:= 1 STEP 1 UNTIL 10 DO
  BEGIN CASE I OF
             K:= T1;
             K:= T2;
              :
             K:= T10;
           ENDCASE;
    T1:= T1+ 1; … T10:= T10+ 10;
  END;
```

実現法の概観

この最適化法を計算機で実現するのは，内部形式を生成する通常の解析部とコード生成部の中間に 3 連のパスを設けることによって行う（図 18.4 参照）. LOOP CHECKER は for ループについて最適化に向いているかどうかを検討し，後で必要になる情報を生成する．つぎのパスは，不変な演算をループの外に移し，3 番目のパスでは演算強さの軽減処理を行う．

ここで原始言語は ALGOL と仮定する．FORTRAN では，DO ループはより単純なので最適化は実際一層容易である．例題として

(18.2.1)　`FOR I := A STEP B UNTIL C DO` 〈実行文〉

を用いる．A, B, C は任意の算術式．さらに，B の値はループ実行中つねに正でかつ不変であると仮定する．これは説明を明快にするためである．B が負の場合への拡張は極めて容易である．

最後に，基本ブロック内での畳み込みと冗長な演算の除去は不変な演算処理の直前または最中に行われることを仮定する．

主たる関心は，単純変数と値呼びの仮引数である．そのための術語を作っておくと便利である：

(18.2.2) **定義**　SFVI (SFVR) とは，**単純変数** (simple variable) または **値呼びの仮引数** (simple formal parameter calbed by value) で INTEGER 型 (REAL 型) のものをいう．

ループの内部表現

演算は（最適化が行われないとして）生成されるべき順序で現れると仮定す

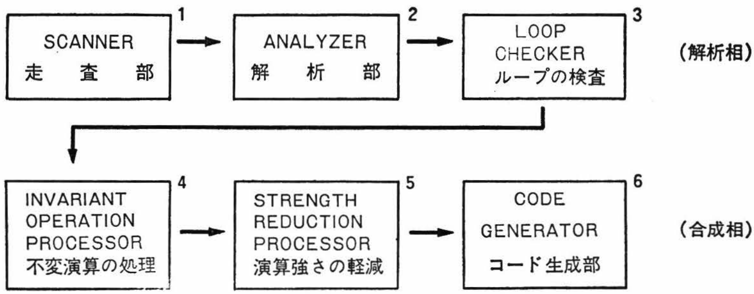


図 18.4　ループを中程度に最適化するコンパイラ

る．演算は，3-組でも 4-組でも，あるいは自分で作った形式で表されてもよい．ループ中の個々の要素が現れた順序で，ループの開始，I, A, B, C, 実行文，ループの終りを認識できることだけが必要である．最適化が行われなければ，ループに対してつぎのようなプログラムと等価なコードを生成することになろう：

```
(18.2.3)    INIT : I := A;
            TEST : IF I > C THEN GO TO OVER;
            LOOP : 〈ループの本体〉;
            INCR : I := I + B;  GO TO TEST;
            OVER :
```

最適化操作により変換されて

```
(18.2.4)    INIT : I := A; “初期設定用の諸演算”;
            TEST : IF I > C THEN GO TO OVER;
            LOOP : 〈変更されたループの本体〉
            INCR : I := I + B; “増分用の諸演算”;
                   GO TO TEST
            OVER :
```

となる．すなわち，ループ本体の実行前に行うべき初期設定用諸演算とループ

の制御変数が増加するたびに行う増分用の諸演算が挿入されている（できれば，I：＝A の直後にもう一つの "IF I＞C THEN GO TO OVER" を挿入して，ループの本体を実行しなくてよい時は，初期設定用演算を省略するようにしたい．初期設定用演算では〈ループの本体〉で用いられる中間結果用変数の値を変えるだけなので，これは必要ないであろう）．

INIT と名札を立てた行は INIT（初期設定）ブロックと称する．INCR と名札を立てた行は INCR（増分）ブロックと称する．ともに基本ブロック——演算は順に実行され，入口と出口が1個づつしかないもの——である．

最適化処理によって新しいブロックが生成され，〈ループ本体〉が変更を受けるので，基本ブロックを節点とするグラフを用いてプログラムを表し，ブロックの変更や生成を容易に行ってみたい気がする．しかしながら，18.3 節に記述する最適化を行おうとするのでなければ，これは必要ないし，不必要なブックキーピングが増えるだけである．つぎの表現の方がもっと適している．

原始プログラムの演算は，表現形式が 3-組であれ，4-組であれ，ポーランド記法であれ，あるいは自分で作ったものであれ，その順にたくわえられている．最適化処理を施すにつれて，3種の個別の表が組立てられてゆく．すなわち各ループごとに，INIT ブロック中の演算を保持しておく表，各ループごとに INCR ブロック中の演算を保持しておく表，および各ループの情報——ループの制御変数，そのループの INIT ブロック，INCR ブロックへのポインタなど——を入れておくループ表である．もちろんコード生成部は，内部形式プログラムの主要部と INIT ブロックと INCR ブロックとに対するコードを正しい順序で生成しなければならない．

あるループの INIT ブロックは，実はそのループの一部ではないことに注意しよう．これは一度しか実行されない．したがってこれは，これを囲むループに属するものとして最適化の対象となる．このことは，ある演算は，入れ子になったいくつかのループを次第次第に外側のループへと移されて，全てのループの外に出てしまうこともありうることを意味している．

INIT ブロックや INCR ブロックの表現についてはこれ以上扱わないことにする．これは主としてコンパイラの構成の問題であり機械ごとに変る可能性が強い．

ループの制限

不変な演算を移すためには，ループ内で変化する SFVI と SFVR を全て

知らなければならない．ループの制御変数 I を含む演算の強さを減ずるにはつぎのような制限を加えることが必要である．

(18.2.5)　　I は〈ループの本体〉内で変化しない．

(18.2.6)　　増分の式 B 内に現れる変数はいずれも〈ループの本体〉内で変化しない．

演算強さの軽減が不可能な場合でも，不変な演算を移動させることができることに注意．しかしながら，この議論では，両者はともに行うか，ともに行わないかのどちらかとする．

FORTRAN では，DO ループが簡単なので条件 (18.2.5) と (18.2.6) はつねに成立する．不幸にして，FORTRAN でも〈ループの本体〉内でどの変数が変化するのか確かめることは不可能である．主たる理由は，ループの本体内で呼出される手続きが，変数の値を変えてしまう可能性にある．したがって，つぎのステップとしては容易にそして正確に検査できるもっと厳密な条件を課すことである．もちろん，できるだけ多くのループを最適化したいという気持を念頭においてのことであるが．

以下の条件は，2種の最適化を遂行するために満たさねばならぬものである：

(18.2.7)　　I，B，C および〈ループの本体〉中で呼出しを許されるのは，呼出しによってどの SFVI や SFVR が変更されるかをコンパイラが知っている手続き，関数，名前呼びの仮引数のみである．

この条件は，標準関数や，入出力文およびコンパイラが，注意して解析した手続き類を呼出すことを許している．FORTRAN では，COMMON 領域に含まれる全ての SFVI および SFVR と実引数はサブルーチによって，必ず変更されるものと仮定しておけば，全てのサブルーチン呼出しを許すことができる．

以下の3条件は，演算強さの軽減を行うために満たされねばならない．

(18.2.8)　　ループの制御変数は SFVI でなければならない．

(18.2.9)　　増分の式 B は整数型でありループの制御変数 I を含むことはできない．式中には定数，SFVI および SFVR 以外が現れてはならない．

(18.2.10)　　ループには，I または増分の式 B に含まれる変数への代入

(代入文，入力文，手続き呼出しあるいは，仮引数呼出し）が含まれてはならない．

本節で述べる最適化を行うためには条件 (18.2.7) から (18.2.10) が満足されなければならない．

ループ表

LOOP CHECKER は，各ループ内で変化を受ける SFVI と SFVR の一覧表を作成する．この一覧表を作成するのに，3通りの方法が頭に浮かぶ．第一はループ内で変化する SFVI と SFVR のエントリを記号表から引出して一覧表を作る方法である．この一覧表はおそらく，番地の連続したものではなく，鎖で連結したリストの形式になるだろう．

第二の方法は，各 SFVI と SFVR に番号を割付け $n \times m$ ブール行列を作るものである．ここで n は SEVI と SFVR の個数で，m はループの個数である．i 番目の変数がループ j で変化するならば $C[i, j]=1$ となる．

第三の方法は，ここで用いるもので，ループ構造を持った表を作る．この表は 9.5 節に述べたブロック構造を持った記号表によく似たものである．ここでは，LOOP 表と称せられるループ用の表が必要で，この表の i 番目の要素は，プログラム中の i 番目のループの記述で，番号は FOR が現れた順番に付ける．これと一しょに，各ループ中で実際に変化する SFVI と SFVR の一覧を含んだ SFV 表が必要である．LOOP 表の 1 要素は下記の 5 欄から成っている：

LOOPVAR	SURRLOOP	FIRSTV	LASTV	BLIST
ループの制御変数	このループを囲むループの数	ループのSFV一覧表中の第1要素へのポインタ	ループのSFV一覧表中の最後の要素へのポインタ	増分の式中のSFVI 一覧表へのポインタ

上の表にある変数は全て記号表の対応するエントリへのポインタによって表すことができる．LOOPVAR 欄はループ中で最適化が可能かどうかを示すために用いることができる．ゼロということは最適化できないことである．理由は後で明らかになるが，LOOP 表のゼロ番目の要素に (0, 0, 0, 0, 0) という値が入っているからである．

```
FOR i:=1 STEP 1 UNTIL N DO
  BEGIN L1:=…; …
    FOR j:=1 STEP 1 UNTIL N DO
      BEGIN L2:=…; …END;

    FOR k:=1 STEP 1 UNTIL N DO
      BEGIN L3:=…; …END;

    L4:=…;
  END;
FOR l:=1 STEP 1 UNTIL N DO
  BEGIN L5:=…; … END;
```

(a) **プログラムの一部**

(1)	i	0	1	6	0
(2)	j	1	3	3	0
(3)	k	1	5	5	0
(4)	l	0	7	7	0

(b) **LOOP 表**

(1)	L1
(2)	j
(3)	L2
(4)	k
(5)	L3
(6)	L4
(7)	L5

(c) **SFV 表**

図18.5　LOOP 表の説明

FIRSTV 欄と LASTV 欄は，あるループの SFV の最初と最後の要素をそれぞれ指している．これは 9.5 節に述べたブロック構造の場合に似ているが，大きな違いがあって，事態はずっと容易である．ブロック構造の場合は，別のブロックに対する表は，別個に保持されねばならなかった．ここでは，表は入れ子状に保持できる．すなわち，もしあるループの中で変数が変化すると，それはそれを囲む全てのループ内で変化したことになる．したがって，あるループに対する SFV のエントリというのは，その内部にある全てのループの SFV のエントリを包含している．

図18.5 はこれの説明である．このプログラム片にはループが四つあり，ループに対応する諸表が与えてある．

LOOP 表の要素は，各ループに対するそれぞれの INIT ブロックと INCR ブロックを記述する欄も持っていなければならないが，ここでは簡単のため省略した．

LOOP CHECKER

このパスは，原始プログラムの演算を，順に処理してループが最適化に向いているかどうか検査する．さらに，LOOP 表と SFV 表を組立てる．ここではつぎのような変数と手続きが使用される：

1. OPENLOOP. 整数型変数，処理している最も内側のループの番号を持つ．初期値は 0.

2. LASTLOOP. 整数型変数，これまでに検出したループの個数を持つ．初期値は 0.
3. OPTLOOP. 整数型変数，処理中の最も内側のループ（ただしまだいずれの最適化可能条件をも犯していないもの）の番号を持つ．
4. CHECK(P). この手続きの引数 P は記号表のエントリへのポインタ．CHECK(P) が呼出されるのは，P が指す変数への代入が行われる時である．この変数が SFVI または SFVR であれば，現在処理中のループは全て（OPTLOOP と各表を利用して）条件 (18.2.10) を検査し，条件の侵犯が検知されれば，そのループは最適化の対象とならない．つぎにこの変数はループ OPTLOOP 用の SFV 表に置かれる．
5. SURROUNDOPT. あるループが最適化の対象からはずれると，OPTLOOP を変更する必要が起り得る．この手続き SURROUNDOPT が，ループ OPTLOOP から始めてそれを囲むブロックの鎖をたどってまだ最適化が可能なループが見つかるまで進み，変更を行う．

OPENLOOP と LASTLOOP は 9.5 節でブロック構造を持った記号表を組立てる際に用いた CURRBL と LASTBL と同様なものであることに注意．

さてここで内部形式プログラムの個々の部分に必要とされる処理を簡単に述べよう：

1. FOR 〈変数〉:= A が処理される時には，代入が行われるので CHECK (〈変数〉) を呼出す．新しいループに対して LOOP 表に適当な要素を追加する．〈変数〉:= A に対する演算をこのループの INIT ブロックのところに移動する．もし〈変数〉が SFVI ならば，OPTLOOP を更新して新しいループ番号を設定する．さもなければ，新しいループの LOOPVAR 欄に 0 を書き込む．これはループの最適化ができないことを示す．
2. 増分の式 B を走査する時は，このループに対する BLIST に各変数を加える．いずれかの変数がこのループの制御変数であったり，SFVI や SFVR でなかったりすると，ループは最適化ができず SURROUNDOPT を呼出す．B が整数型でなければ，ループの最適化は不能とし SURROUNDOPT を呼出す．
3. 代入文〈変数〉:=，や READ (〈変数〉) のところでは，CHECK (〈変数〉) を呼出す．

4. 標準的でない手続きや関数はどんなものでもそれを呼出しているところで，あるいは，名前呼びの引数（配列ではない）を参照しているところで，現在処理中のループを全て最適化不能とし SURROUNDOPT を呼出す.
5. あるループの終りのところで，もしこのループがなお最適化可能となっていれば，そのループに対する SFV 表に対して BLIST を検査する．もし両方の表に入っている変数があれば，このループは最適化不能とする．これを囲むループを指すように OPENLOOP と OPTLOOP の値を改め SURROUNDLOOP を呼出す.

ステップ2.からステップ4.の実行が必要なのは最適化可能なループが処理中である，すなわち OPTLOOP ≠ 0 の時だけである.

ループの最適化可能性を検定する比較的複雑な過程を，より基本的で単純な数ステップに分割したことに注意．ステップ4.の検査は1個のルーチンの中だけで行うべきである．これは内部形式プログラムを走査し，記号をこのパスの他の部分へ渡すルーチンの中で行うのが最も容易であろう．このルーチンがある名前を走査しており，最適化可能ループが処理中のときは，この名前が手続き名か関数名かあるいは名前呼びの仮引数であれば，現在処理中のループは全て最適化不可能とすべきである.

INVARIANT OPERATION PROCESSOR（不変演算の処理）

あるループ内である演算が不変であるとは，その全ての被演算子がこのループ内で変化するどんな変数にも依存しないことをいう．このような演算は，このループから外に出して INIT ブロックに入れることができることが多い．ここでは ＋，－，＊，ROUND（丸め），CVRI（変換）のような演算だけを扱う．代入演算は考察しない．この演算は，プログラム解析が必要でこれまでの準備だけでは間に合わないからである．中間結果用変数 T*i* は（これまでと同じように）値の代入は一度かぎりとし，原始プログラム上の変数とは区別されていると仮定する.

このパスは，内部形式プログラムの演算を順にたどって，LOOP 表と SFV 表を用いて不変式を検出する．移動できそうな演算に出会うと，つぎのステップを踏む：

1. 例えば，その演算が（＊ A, B, T1）とする．A あるいは B がこのル

ープ内で不変でなければ（これは現在処理中の最適化可能ループに対する SFV 表を検査すればよい），以下のステップは行わない．

2. 演算（* A, B, T2）がすでに INIT ブロックにあれば，ステップ 4. に進む．さもなければステップ 3. へ．
3. この演算を INIT ブロックに加えてステップ 5. へ．
4. ループ内でのこれ以後 T1 を参照するものは T2 へ変更する．
5. ループからこの演算を削除する．このループ内で変化する変数の一覧表から T1 を削除する（ただしこれを囲むループの表からは削除しない）．

これは図 18.6 **a** と図 18.6 **b** に示した．ループ内で変化する変数の一覧表から T1 を削除することに注意．いつもやるように，各中間結果用変数は一度かぎりしか変化しないと仮定している．これによって以後に続く T1 を含む演算も外に移すことができるようになる．何度も変化するような変数は，何らかの方法でしるしをつけておく必要がある．

最適化可能なループに新しく出会うと，新しい中間結果用変数 T0 に対する代入文 T0 := B をそのループの INIT ブロックに挿入することができる（B が定数や単純変数でない場合）．これが実行できるのは，ループの条件のために，B 中の全ての演算が不変だからである．このブロックの中で B を参照するところはどれも T0 を参照するものとして扱われる．

ループの終りに到達して，このパスが終ると，以下に述べるように，演算のいくつかは INIT ブロックに移されている．INIT ブロックは今処理の終ったループの外側にあって，これを囲むループの内側にある．そこで，これらの演算はさらに外側に移せるかもしれないので，このパスは，普通の処理の前にこの INIT ブロックをこれを囲むループの一部として処理する．

演算の強さを軽減するパス

最適化の最終パスは，各ループの演算を順にたどる．（* I, K, T1）あるいは（* K, I, T1）のような形式の演算（K はループ内で不変とする）を発見すると，これを NO-OP 演算（なにもしない）とし，つぎのステップを踏む：

1. 演算（* I, K, T3）がすでに INIT ブロックにあれば，すでに同じ演算の強さの軽減を済ませている．以後この〈ループの本体〉内で T1 を参照しているものは全て T3 の参照に変え，以下のステップは省略する．
2. 演算（* I, K, T1）を INIT ブロックに加える．

```
INIT:I:=A



TEST:test I>N
LOOP:T1:=I*d2
     T2:=T1+J
     T3:=J*d2
     T4:=T3+I
     T5:=A[T2]
        +A[T4]
     A[T2]:=T5
INCR: I:=I+1

     GOTO TEST
OVER:
```

(a) もとのプログラム

```
INIT:I:=A
  T3:=J*d2


TEST:test I>N
LOOP:T1:=I*d2
     T2:=T1+J

     T4:=T3+I
     T5:=A[T2]
       +A[T4]
     A[T2]:=T5
INCR: I:=I+1

     GOTO TEST
OVER:
```

(b) 不変演算処理

```
INIT:I:=A
     T3:=J*d2
     T1:=I*d2
     TI:=1*d2
TEST:test I>N
LOOP:
     T2:=T1+J

     T4:=T3+I
     T5:=A[T2]
       +A[T4]
     A[T2]:=T5
INCR:I:=I+1
     T1:=T1+TI
     GOTO TEST
OVER:
```

(c) 演算強さの軽減

FOR I:=1 STEP 1 UNTIL N DO BEGIN A[I, J]:=A[I, J]+A[J, I] END

図18.6　2通りの最適化

3. もし演算（* B, K, T2）または（* K, B, T2）が INIT ブロック中になければ，中間結果用変数 T2 を新しく生成し，演算（* B, K, T2）を INIT ブロックに加える（このような演算は，先の INVARIANT OPERATION PROCESSOR のパスでそこに置かれている可能性がある）．
4. 演算（+ T1, T2, T3）を INCR ブロックに加える．

ここで各中間結果用変数 T*i* は値の代入は一度かぎりと仮定していた．中間結果 T1 はここでは，2ヶ所——INIT ブロックと INCR ブロック——で値を代入されている．しかしながら，これら2ヶ所の代入文はどちらもこれ以上最適化されることはない．INCR ブロックにあるものは，このブロックはもはや最適化の対象とならないので最適化されない．INIT ブロックにある演算（* I, K, T1）はこの代入が今生じているループの中で I が変化しているのでこれ以上最適化できない．

二つの技法の検討

図18.6 b は図18.6 a のループから不変な演算を移した結果を示している．**図18.6 c** は，それに演算強さの軽減を施したものを示している．不変演算処理が先に適用されていることが重要である．例えば（X + Y）* I を考えよう．

ただし X と Y は不変とする．内部形式で，これは

(1) ＋ X, Y, T1　　　(2) ＊ T1, I, T2

と表せる．乗算の強さは T1 がループ内で不変でないので軽減することはできない．しかし，もし最初の 4-組が，不変演算処理のパスで INIT ブロックに移されていると，T1 はループ内で不変となる．

これはまた ＋ や ＊ を 2 項演算とするかわりに n 項演算として処理すると都合がよい．この場合，被演算子はつぎの規則に従って並べるとよい：

1. この演算のあるループの制御変数．
2. ループ内で不変な被演算子．
3. ループ内で不変でない被演算子．

それぞれの分類の中では，前節で述べたように並べたらよい．この順番はどのループにこの演算が存在するかによって変ってくる．例として，Y1＊X1＊I＊X2＊Y2 を考えよう．ここで Y1 と Y2 は不変ではなく X1 と X2 が不変とする．これを 4-組で表すと

(1) ＊ Y1, X1, T1　　　(3) ＊ T2, X2, T3
(2) ＊ T1, I, T2　　　(4) ＊ T3, Y2, T4

となりこれに対して最適化はありえない．しかしながら，これを ＊Y1, X1, I, X2, Y2, T1 と考えると（結果は T1 に置かれる）並べ換えを行って

＊ I, X1, X2, Y1, Y2, T1

となり，＊X1, X2, T2 を除去して ＊I, T2, Y1, Y2, T1 が得られる．最後に，乗算 I＊T2 の強さを減じることができる．

“かくれた” 最適化として重要なものの一つに添字付変数に関するものがある．8.4 節で述べたように，添字付変数 X[$i, j, \cdots, p, q$] の評価には，VARPART の評価が必要である．これは

$$((\cdots(((i*d_2)+j)*d_3)+\cdots)+p)*d_n+q$$

で与えられる．もし i がループの制御変数であれば，$i*d_2$ の強さを減らすことができる．最適化のためには，

$$i*d_2*\cdots*d_n+j*d_3*\cdots*d_n+\cdots+p*d_n+q$$

を用いて VARPART を計算する方が都合が良い．何となればこの式の計算は n 個の個別の項に分割され，それぞれが独立に最適化の対象となりうるからである．

3-組を用いる場合，演算を基本ブロックから移動するときは用心しなければならない．3-組への参照は，その 3-組の特定の実行結果を参照している．基本ブロック内では，このような参照はつねに，この 3-組が最後に実行された結果に対するものである．基本ブロック内に

(1)　(+ A, B)　　　　(2)　(+ (1), C)

があり，最初の 3-組はループの外に移動できると仮定しよう．すると2番目の 3-組の中で (1) を参照することが，移動された 3-組の結果を参照しているという事実を保持しておくことは難かしい．最初の 3-組を移動する際に，新しい中間結果 T を生成し，2番目の 3-組を (2)(* T, C) と変更し，つぎの演算を INIT ブロックに置くのが一層好都合である：

(k)　(* A, B)　　　　($k+1$)　(:= (k), T)

演算強さの軽減法の拡張

最適化技法は (I + K)*X のような式ではうまく働かない．ただし，I はループの制御変数で K と X をループ内で不変とする．これを改善する方法は一つには I を含む部分式を識別して，その形を変えるもので，この場合 I*X + K*X を作る．すると最適化はより効果的になる．別の解決法は演算の強さを二つの方向に拡大するものであるが，どちらも実際には最適化の複雑さを増すようなものではない．

第一のものは，加算 T1 := I + K の強さをつぎのようにして減らすものである：

1. T1 := I + K を INIT ブロックに置く，
2. T1 := T1 + B を INCR ブロックに置く (I が B だけ変化し，したがって T1 も同じ)．
3. 演算 T1 := I + K を削除する．

第二の方向は，ほかに再帰的に定義されている変数も演算強さの軽減に加えるものである．これ以上原始プログラムを解析しないでも，この候補になりそうなものは，上の T1 のような中間結果用変数である．これは，それ以前に行

われた強さの軽減によって再帰的に値が定義されるようになっていた.

これを以下で説明しよう. ループの中にある D := (I * X + Y) * Z に対する 4-組を段階的に最適化してゆく過程を示す (INIT ブロック, INCR ブロック, LOOP ブロックのみを示す). 3 個の演算子は, その都度新しく生成されて再帰的に定義される変数を伴って全て最適化されている.

(a) I*X の軽減	(b) T1+Y の軽減	(c) T2*Z の軽減	(d) 最終結果
INIT:I:=A	INIT:I:=A T1:=I*X T4:=B*X	INIT:I:=A T1:=I*X T4:=B*X T2:=T1+Y	INIT:I:=A T1:=I*X T4:=B*X T2:=T1+Y T3:=T2*Z T5:=T4*Z
LOOP:T1:=I*X T2:=T1+Y T3:=T2*Z D:=T3	LOOP: T2:=T1+Y T3:=T2*Z D:=T3	LOOP: T3:=T2*Z D:=T3	LOOP: D:=T3
INCR:I:=I+B	INCR:I:=I+B T1:=T1+T4	INCR:I:=I+B T1:=T1+T4 T2:=T2+T4	INCR:I:=I+B T1:=T1+T4 T2:=T2+T4 T3:=T3+T5

INCR ブロックに入った 4-組の多くは最適化が進むにつれて無用になり削除した方がよくなることに注意. 上の (d) 最終結果で, T1 と T2 が参照されることは全くないので, T1 := T1 + T4 と T2 := T2 + T4 は削除できる.

より少ないパスでの最適化

図 18.4 は, 3 連のパスを追加して最適化を行う様子を示している. 普通はもとの内部形式プログラムを生成する意味付けルーチンと LOOP CHECKER とを合併することが可能である. 場合によっては最適化のパス全てをコード生成パスと合併することもできる. 3 連のパスがあることによって必要となるたくさんのこまごましたブックキーピングをこれによって省略しコンパイル時間を短縮することができる. 最適化とコード生成との合併をどのように行うかに

ついて簡単に述べてみよう.

(18.2.4) に示したように INIT ブロックはループの手続きの前に置かれるものであるが，このループが全部処理されるまでは完全なものにならない. この問題に対処するために，これと等価なつぎのようなコードを生成する：

```
                 GO TO INIT;
(18.2.11)  TEST: IF I > C THEN GO TO OVER
           LOOP: 〈変更されたループの本体〉
           INCR: I := I + B; “増分用の諸演算”; GO TO TEST;
           INIT: I := A; “初期設定用の諸演算”; GO TO TEST;
           OVER:
```

このプログラムには，分岐命令が二つ増えているが，これによって〈ループの本体〉を最適化しそれに対するコードを同時に生成することができるようになる. そしてその後に INIT ブロックと INCR ブロックに対するコードを生成する.

内部形式プログラムは，それぞれ別個の三つの入口から入力されて，これからコードが生成される. 三つの入口とは，中心となる原始プログラム用の表と，INIT ブロックと INCR ブロックである. これは単に，現在処理中のプログラムがどの入口から入って来ているかを憶えておくだけのことである. さらに，最適化を行うか否かは，適当な大域的変数を設定して自由に切換えることができるようにしておかなければならない.

コード生成部はまず最適化を行わないことにして処理を開始する（ただし，畳み込みと冗長演算の除去はこの限りでない). 入力の入口はまず中心となる原始プログラム用の表にしておく. 演算は，現れる順序に，演算の型に応じて処理される. これは下に示すとおり. 演算の分類は優先順である.

1. 畳み込み可能な演算：畳み込む.
2. 冗長な演算：先に述べたようにして除去する.
3. 最適化可能な不変演算：最適化の実施——現在のループの INIT ブロックにその演算を移動する.
4. 演算強さの軽減が可能な演算：最適化を実施する.
5. ループ演算の開始：新しいループを開始するためのブックキーピングを行う. I := A; TO:= B; を INIT ブロックに置く. I:= I + TO を INCR ブロックに置く. 名札 INIT を立て，そこに分岐するコード

を生成する．名札 TEST と名札 OVER を立て

TEST: IF I > C THEN GO TO OVER

に対するコードを生成する．名札 LOOP の生成．“つぎの演算へのポインタ”を〈ループの本体〉の最初の演算を指すように改める．ループの最適化が可能であれば，最適化スイッチをオンにする．

6. 〈ループの本体〉の終り：最適化スイッチを切る．入力の入口をこのループの INCR ブロックに切換える．
7. INCR ブロックの終り：GO TO TEST 命令を生成し，現在のループに対する INCR ブロックを削除する．このブロックに対するコードは生成されたばかりである．名札 INIT を立てる．これを囲むループを新しい処理のループとし，それが最適化可能ならば，最適化スイッチをオンにする（したがってこの INIT ブロックは最適化の対象となる)．入力の入口を今処理の終ったループに対する INIT ブロックに切換える．
8. INIT ブロックの終り：GO TO TEST を生成する．名札 OVER を立てる．今処理を終えたばかりの INIT ブロックを削除する．入力の入口を，中心となる内部形式プログラムに切換え，先ほどのつづきの演算から処理を開始する．
9. 分岐先となる名札または演算：畳み込み用および冗長な演算用の表を空にする．ここから新しく基本ブロックが始まる．
10. その他の演算：それに対するコードを生成する．

18.3 より効果的な最適化

ここで述べる最適化手続きには内部形式プログラムを有向グラフに直す必要がある．ここで基本ブロックが節点となる．このことについては，11.5 節をもう一度のぞいてみるのがよいだろう．内部形式自体は，4-組でも，3-組でも，間接 3-組でもよく，ただ細かい点が違っているだけである．われわれは 4-組を用いよう．ここでは，計算機上での実現法を詳しく述べるのではなく，使用している最適化の種類とそれを行う順序を述べる．Allen (69) による優れた論文（本節での考え方の多くはこの論文に依っている）と Lowry と Medlock (69) を参照されたい．

まず，ループはより基本的な要素——代入文，判断，分岐など——の系列に分解されているものとする．例えば，内部形式プログラムのレベルでは，下記

の二つのプログラム片の間には，全く差がない．

```
FOR I:=1                          I:=1;
  STEP 1 UNTEL N DO         LOOP: IF I>N THEN GO TO OVER;
    A[I]:=A[I]+1;                 A[I]:=A[I]+1;
                                  I:=I+1; GO TO LOOP;
                            OVER:
```

最適化の第 1 段は，プログラム・グラフの部分グラフを見出すものである．これは後で定義するが**領域**（region）と称せられる．普通の表記法で書いたループに対する部分グラフと上のように判断と分岐を用いて書いたこれに等価なループに対する部分グラフとはともに領域を形成する．しかしながら，領域の概念を用いると for ループの**内部**にある程度まで最適化が可能な他の部分グラフを見出すことができるようになる．

領域上で最初に行われる 4 種の最適化は，つぎの順による：

1. 基本ブロック内での畳み込み．
2. 基本ブロック内での冗長な演算の除去．
3. 不変な演算の領域外への移動．
4. 領域内における演算強さの軽減．

はじめの二つは 18.1 節に述べたように厳密に行なわれる．これらについてはこれ以上議論しない．あとの二つは，お気付きのように，for ループではなく領域に適用されている．これは前節に述べた同じ二つの技法よりもっと一般的なものである．演算強さの軽減を終えた後には，その領域はさらにつぎのような修正の対象となる：

5. 判断の置換え：プログラマが書いた判断命令を別のものと置換えることによって領域内にある別の演算を削除できるようになる．
6. “死んだ”変数と代入文の除去，およびそれらが依存する演算の除去．

領域の内部（基本ブロックの内部ではなく）では冗長な演算の除去も行うことができようが，これは時間を消費する割には効果があがらない．

領域と領域リスト

有向グラフの**強連結領域**（strongly connected region）とは，部分グラフであってそれに含まれる任意の節点からそれに含まれる他の全ての節点への道が

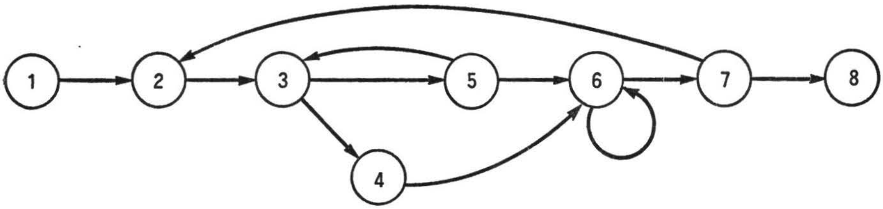


図18.7 有向グラフの例

存在するものをいう. 図18.7には5個の強連結領域がある：(6), (3, 5), (2, 3, 5, 6, 7), (2, 3, 4, 6, 7), および (2, 3, 4, 5, 6, 7). 部分グラフ (3, 5, 6) は強連結領域ではない. 節点6から3または5に行く道がないからである. 同様にして, 節点1や8は, どの強連結領域にも含まれない.

図18.8は, 約束ごとの for ループ†がループ外への分岐を伴っている例である. 初期設定 I := 1 を除いて, 完全なループが一つの領域を形成していることに注意 (図のブロック 2, 3, 4, 5, 6).

```
AGAIN:
FOR I:=1 STEP 1 UNTIL n DO
  BEGIN · · ·
    IF X THEN GOTO AGAIN;
        · · ·
  END;
```

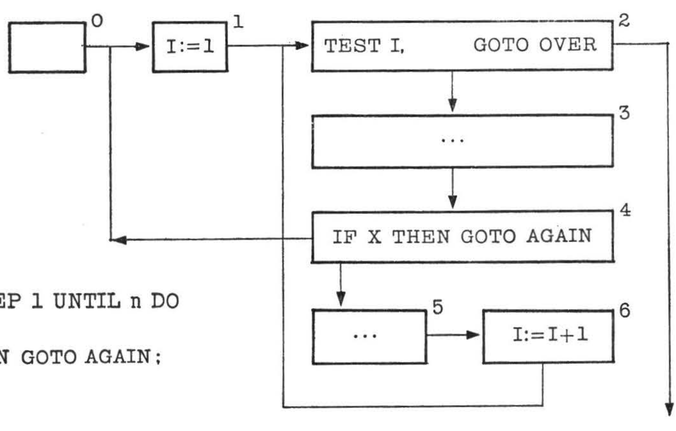


図18.8 分岐を内部に含んだループとそのプログラム・グラフ

† 訳注 ループを記述するには, ALGOL の繰返し文 (for ループ) や, FORTRAN の DO 文のようにそれを書けばループと約束しているもののほかに, 例えば IF 文と GO TO 文を組合せても可能である. ここでは前者を '約束ごとのループ' と称する.

領域の**入口** (entry) ブロックとは，この領域の外部から直接矢印を向けられているブロックをいう．図18.7では，領域 (6) は，それ自身入口ブロックである．領域 (3, 5) ではブロック3が入口ブロックである．

領域の**先行** (predecessor) ブロックとは，その領域の入口ブロックに領域の外部から直接矢印を向けているブロックをいう．図18.7では，領域 (6) は二つの先行ブロックを持つ：ブロック4および5．領域 (3, 5) は先行ブロックが一つで，ブロック2である．**後続** (successor) ブロックも同様に定義する．

領域の**合流** (articulation) ブロックとは，先行ブロックから，領域を通って，後続ブロックへ至る全ての道の上に現れるブロックをいう．図18.7では，ブロック6は領域 (6) の合流ブロックであり，ブロック3が領域 (3, 5)のそれであり，ブロック2, 3, 6, 7が領域 (2, 3, 4, 5, 6, 7) の合流ブロックである．ブロック4と5は，この領域の合流ブロックではない．

最適化の前に，基本ブロックを節点とするプログラム・グラフの形式で，原始プログラムを，解析し，強連結領域 $R_1, R_2, \cdots, R_n$ からなる一覧表 R をつぎのような基準で作成する必要がある．

1. $i=j$ ならば，$R_i=R_j$
2. $i<j$ ならば，R_i と R_j は共通なブロックを持たないか，R_i の全てのブロックが R_j に含まれるかのいずれかである．

R に列挙されている領域はそれぞれ互に約束ごとのループが持っていると同じ構造を持っている．すなわち，入れ子になっているか，共通なブロックを持たず，したがって“並行”であるかである．一部分だけが重なるということはない．領域 $R_1, R_2, \cdots, R_n$ はこの順に最適化処理を施される．もちろん18.2節で各ループの最適化可能性を検査したように各領域 R_i についてその可能性を検査しなければならない．これについては再び言及しない．

図18.7のグラフに対して，つぎのような領域一覧表を展開できよう：

$$R_1=(6),\quad R_2=(3,5),\quad R_3=(2,3,5,6,7),\quad \text{および}$$
$$R_4=(2,3,4,5,6,7).$$

図18.8のプログラムには，二つの強連結領域 (1, 2, 3, 4) と (2, 3, 4, 5, 6) がある．両者を一しょに領域一覧表にのせることはできない．上の条件2を犯すことになるからである．ではどちらを一覧表にのせたらよいか．最も効

果的な最適化は，繰返し回数の多い領域——その中でプログラムが最もよく実行される領域——から演算を取除いたり，強さを軽減したりすることである．われわれの持っているプログラミングの知識から，約束ごとのループは実行回数が多いと推定でき，そのようなループを領域一覧表にのせるのが理に適っているようである．

約束ごとのループに対する部分グラフが R に列挙されていることはどのようにして確かめることができるのか．もちろん，方法の一つは，それを何らかの方法で内部形式プログラムの中に作っておくことである．こうはしたくない場合もあるので，こういうループを検出する別の方法を導入しよう．

約束ごとのループに対する領域は，先行ブロックをただ一つしか持たない．すなわちループの制御変数の初期設定を含んだブロックである．そしてこの領域の入口ブロックは，この先行ブロックから見ると唯一の後続ブロックである．この性質はどの領域を一覧表にのせるかを決める際に大変役に立つ．何となれば，もし二つの領域が重複した部分を持っているが，一方が他方を含んでいない場合，ただ一つの先行ブロックとただ一つの入口ブロックをともに持つような領域が高々一つあるということを主張できるからである（証明はしない）．したがって，もし二つの領域が重なっていれば，入口ブロックを一つと先行ブロックを一つ持つ領域を選べばよい．

図18.9 のグラフを考えよう．これには三つの領域 (1, 2)，(2, 3) および (1, 2, 3) がある．上の規則を用いれば，領域一覧表としては $R_1 = (2, 3)$，$R_2 = (1, 2, 3)$ が得られる．

データの依存性

各ブロックに対して，その中でどの変数が参照されているか．どの変数が値を代入されるか，どれが代入の前に参照されるかを知る必要がある．このた

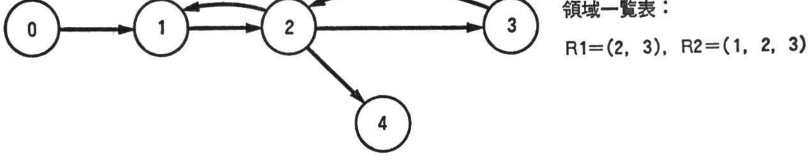


図18.9 プログラム・グラフの例

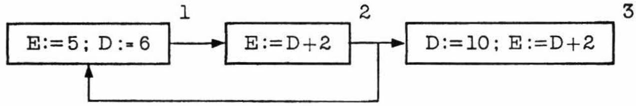


図 18.10　出口で忙しい変数の例

め，原始プログラムの変数と中間結果用変数の各々に番号を割付ける．プログラムの各基本ブロックごとにつぎのような三つの論理型のベクトルを構成する：

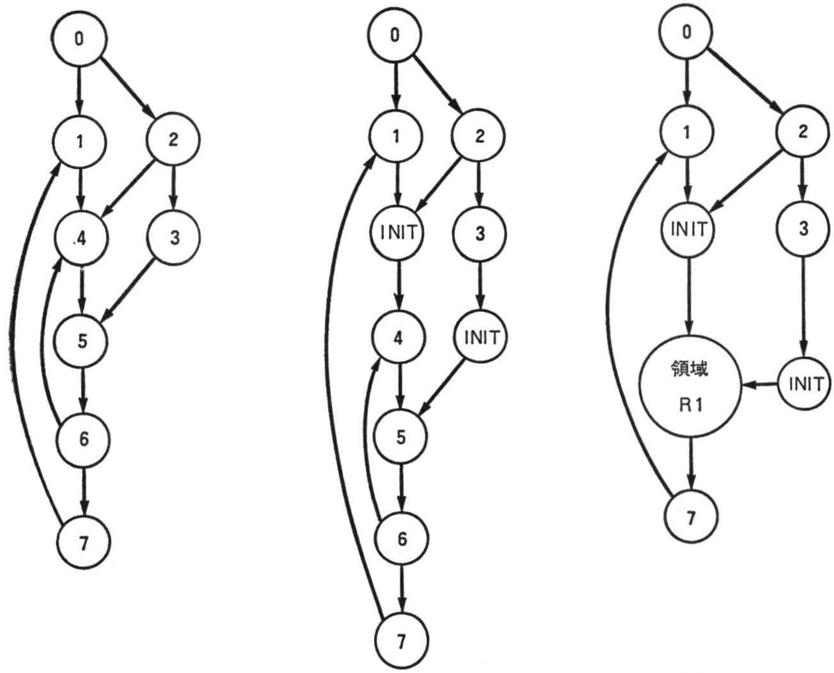
1. REF[i] = 1：このブロックで i 番目の変数が参照されている．
2. ASS[i] = 1：このブロックで i 番目の変数に値が代入されている．
3. RBA[i] = 1：このブロックで i 番目の変数は値が代入される前に参照されている．

変数がブロックの**出口で忙しい** (busy on exit) とは，ブロックの終りで，変数に入っている値が後で参照されることをいう．図 18.10 のプログラム片では，D はブロック 1 の出口で忙しいが，E は忙しくない．領域の出口で忙しいということも同様にして定義する．各変数の"忙しさ"を示すためにもう一つベクトルを用意することもできようが，この情報は，そのブロックの全ての後続ブロック（およびそのまた後続ブロック，…）の REF, ASS, RBA ベクトルを調べれば得ることができる．

RBA ベクトルの処理は難かしい．このベクトルを使う必要があるのは，代入演算をループの外に移す最適化と死んでいる代入演算を除去する最適化を行う場合のみである．

ある領域に含まれる個々のブロックに関する情報はこの領域自体に関する情報を得るために用いることができる．例えばこの領域でどの変数が用いられている（代入されている）かを決めるには全てのブロックの REF (ASS) ベクトルの"論理和"をとればよい．

プログラムが大きくなると，恐ろしくたくさんのベクトルを保持しなければならなくなる．一時に処理する領域は一つだから，処理中の領域にあるブロックに対するベクトルを保持するだけでよさそうである．これを示そう．



(a)　R1=(4, 5, 6)　　(b)　　(c)

図18.11　領域 R_1 の最適化

最適化計画の大様

ここで，一時に一つの領域を $R_1, R_2, \cdots, R_n$ の順で最適化する．したがって，最も内側にあるループが最初に処理される．最適化の結果，INIT（初期設定）ブロックと呼ばれる，新しいブロックが創出され，これは領域に入る直前に実行されるべきものである．**図 18.11 a** のグラフで領域 $R_1 = (4, 5, 6)$ を考えてみよう．R_1 を最適化すると**図 18.11 b** のグラフが得られる．この領域には入口ブロックが 2 個あるので，図 18.11 b では INIT ブロックの写しが 2 個必要である．ブロック 3 の後続ブロックはただ一つであり，INIT ブロック演算は，ブロック 3 の終りの部分に含めて，余計なブロックが一つできるのをさけることができる．

一たんある領域が最適化されると，二度と最適化される必要はない．したがって，すぐ外側にある領域を最適化する時には，最適化の終った領域は 1 個の

ブロックと見なすことができる（図11.8c 参照）．このブロックは最適化済みのしるしをつけられて，再び処理されないようにする．この領域に対するREF，ASS，RBA ベクトルは，この領域にある個々のブロックに対するベクトルから上述したように容易に得られる．

INIT ブロックは，もちろん，これを囲む領域に含まれることになるので最適化の対象となる．これは最も内側にある領域に含まれる演算が，それを囲む領域によって一段づつ取除かれ得ることを示している．

一般に，ブロック用の三つのベクトルを必要とするのは，現在最適化の対象となっている領域にあるブロックに対してだけである．これらのベクトルは，基本ブロックに対して，畳み込みと冗長な演算の除去を行っている間に組立てることができる．しかしながら，最適化の進行中に，これらのベクトルは演算が移動したり変化したりするにつれて変化することに注意しよう．さらに，新しく中間結果用変数が創出されると，それに番号を割付け，各ブロックに対する三つのベクトルに追加してやる必要がある（これはその記号表のエントリにその変数に関する情報を保持することができることと同等である．すなわち，これら変数の使用法は正確に分かっているので，このようなことが可能なのである）．これは最適化の手続きを議論するところでは述べないが，必須のことである．

一つのプログラムに対する完全な処理過程はつぎのとおりである：

1. $i := 1$ とする．
2. 領域 R_i を最適化のために取出す．INIT ブロック用の場所を取り分けておく．
3. R_i の各種ブロックに対して畳み込みと冗長な演算の除去を行う．同時に，REF，ASS，RBA の各ベクトルを構成する．この領域内で再帰的に定義される SFVI の一覧表を作成する．これらの変数は演算強さの軽減の際に使う可能性がある．前節でループの制御変数に用いた方法．
4. 領域の最適化を行い，INIT ブロックを創出する．これによってブロックベクトルが変化し新しく中間結果用変数が創出される可能性がある．
5. 領域 R_i に対する三つのベクトルを作成する．以後はこれを最適化の必要のないブロックと見なす．INIT ブロックを正しい場所に挿入する（この領域の先行ブロックと入口ブロックの間の全ての道の上に挿入する）．

6. $i := i+1$. もし $i \leq n$ ならば，ステップ2.へ．さもなければステップ7.へ．
7. どの領域にも属さない基本ブロック内で畳み込みと冗長演算の除去を行う．

つぎに，領域の各種の最適化（上のステップ4.）を記述することに進もう．説明は適用の順序にしよう．

不変演算の移動

われわれはすでに ＋，*，CVRI などのような演算について，不変演算をループの外に移すことを議論した．この処理過程は領域の外に演算を移す場合でも同様である．このほか，代入演算も移動できることを示そう．そのための制限は（A := B または A := T，ただし T は中間結果，という代入を考慮して）つぎのようなものである：

1. B（または T）はその領域内で不変でなければならない．
2. この代入が行われるのはその領域の合流ブロックでなければならない．これによって，領域に入ったところでこの代入を実行できることが確かめられる．
3. この領域ではこれ以外に A への代入はない．
4. 入口ブロックからこの代入へ至る任意の道の上に A を参照する演算がない．

このらの制限条項は，この領域の基本ブロックに対する三つのベクトル REF, ASS, RBA を用いて検査することができる．図 18.12 を考えよう．代入文 B := 5 は移動できる．代入文 E := 4 はできない．すなわちこれは，条件3に違反している．代入文 A := 3 も移動できない．これは条件2違反で



図18.12 不変な演算の説明

ある．代入文 D := B は，一たん B := 5 が移動されると，移動可能である．代入文 C := E は条件1を犯しており，代入文 F := 5 は条件4を犯している．

演算強さの軽減

ある領域で再帰的に定義される変数 I を含む演算の強さを軽減するためには，つぎの2条件が成立たねばならない：

1. I は整数型の単純変数であるか値呼びの整数型の仮引数でなければならない (SFVI)．
2. この領域内で，I に対する代入文は全て I := I + D の形式でなければならない．ただし D は，この領域で不変な符号付きの整数型の式である．

I が n 個の代入によって再帰的に定義されるものとする：

I := I + D1;　I := I + D2; ⋯; I := I + Dn;

K が整数型でこの領域で不変とすると，演算 T1 := I*K をつぎのようにして軽減する：

1. T2 を新しい中間結果用変数として，演算 T2 := I*K を INIT ブロックに加える．
2. n 個の中間結果用変数 TINCRi を生成し，TINCRi := Di*K を INIT ブロックに加える．
3. 領域内の演算 T1 := I*K を T1 := T2 と入換える．
4. I := I + Di のつぎに演算 T2 := T2 + TINCRi を挿入する．ただし $i = 1, \cdots, n$.

これにはいくつかの改善が可能である．もし Di = 1 ならば（このような場合は多い），TINCRi は不要であり，T2 := T2 + TINCRi の代りに演算 T2 := T2 + K を挿入すればよい．つぎに，もし K と Di が定数であれば，この演算を畳み込んでその結果を TINCRi の代りに用いることができる．第三に，いくつかの条件が成立てば，新しい中間結果 T2 を使う必要がない．この場合，T2 の現れるところは全て T1 を用い，したがって T1 := T2 を挿入する必要はない．条件というのは，よく生ずるもので，(1) T1 はループ

の中でほかの値を代入されないこと，(2) T1 を使用するところでは全てこの代入に依存すること，(3) T1 はこの領域の出口で忙しくないこと，である．

演算 T1 := I + K の強さの軽減はつぎのようにして行う．

1. T2 を新しい中間結果用変数として新しい演算 T2 := I + K を INIT ブロックに加える．
2. 演算 T1 := I + K を T1 := T2 と入換える．
3. 演算 I := I + Di の後ろに演算 T2 := T2 + Di を挿入する．ただし $i=1, \cdots, n$.

同様に T1 がいくつかの条件を満たせば新しい中間結果用変数 T2 を生成する必要がないことに注意．

演算強さの軽減のために挿入される増分の演算 T2 := T2 + TINCRi と T2 := T2 + Di は，これ以上強さを軽減されないように何らかのしるしをつけておく必要がある．前節では，これは必要なかった．何となれば，この増分の演算は INCR ブロックに入れられ，このブロックはそれ以上最適化されなかったからである．ここでは，INCR ブロックをわざわざ別に設けていない．

T2 は再帰的に定義されるようになり，演算強さの軽減の対象となりうることに注意しよう．しかし増分の演算 T2 := T2 + TINCRi は，軽減してはならない．

この最適化は，本質的には，前節で行ったものと同様であるが，ここでは，再帰的に定義された全ての変数について強さの軽減を認めており，ループの制御変数だけではない．

判定の置換え

図 18.13 c で，I が用いられているのは，再帰的に定義されるところと，I を判定するところだけである．この判定を，何かほかの変数の判定に置換えると，演算 I:= I + 1 を除去できる．この例では，上の判定を T1 または T2 の判定と取換えることができる．T2 の方が適当で，それは演算 T1 := T1 + K1 がすでに冗長でありいずれ除去されるからである．

判定 IF I > C … を判定

```
IF T2 > C2 THEN GO TO OVER
```

に置換えたい．さて，T2 は I を用いると T2 := T1 + K2 = I * K + K2

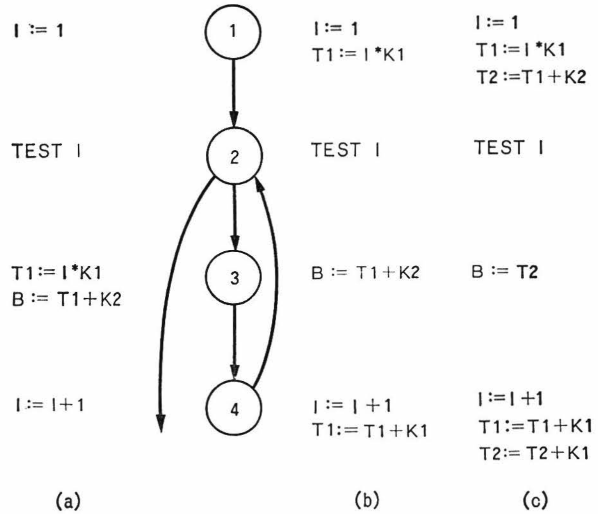


図 18.13　演算の強さの軽減の説明

で定義されている．したがって，もし I > C ならば T2 > C＊K + K で，代入文 C2 := C＊K + K2 を INIT ブロックに加えて C2 を初期設定する．ただし C は領域で不変としていることに注意．さらに I および T2 に対する増分が異符号ならば，判定の不等号は ″>″ から ″<″ に変更しなければならないことも注意しよう．

判定の置換えは，つぎのような条件のもとでのみ行うべきである．

1. 考えている判定が変数 I を含みそれをほかで使用しているのは I := I + K のような再帰的定義においてのみである．I が出口で忙しくてはいけない．
2. この判定は，解析するのが十分単純でなければならない（例えば I > C）．C は，領域で不変でなければならない．
3. ほかに再帰的に定義される変数 T が存在しなければならない．これは究極的には I を用いて演算強さの軽減によって生じたもので，この定義は I を含まねばならない．

この最適化の効果には疑問の余地がある．これによってループの中で加算が

1～2個減るかもしれないが，コンパイル時において I が出口で忙しいか否かを見出すのに要する時間はかなりなものである．もちろん，ALGOL や FORTRAN では，ループの制御変数の値は，ループ脱出後は未定義となるので，この変数は，出口では決して忙しいものとはならない．

死んだ代入文の除去

代入演算が“死んでいる”とは，その代入の結果が用いられることが決してないか，その代入が（I := I + 1 のように）再帰的であり，このような再帰的定義の中だけで用いられていることをいう．このような代入とそれが依存している演算は除去可能である．

死んでいる演算は演算強さの軽減と判定の置換えを行った結果生ずることが多く，図 18.13 に示すようなものである．したがってこの最適化はつねに判定の置き換えを行った後に実施すべきである．明らかに，再帰的に定義されている変数は全て，“死んでいること”を検査すべきである．しかし，領域内に含まれる全ての代入を検査するのは，かえって時間を浪費する．このような代入（A := T を例とする）を検査する大雑把な方法はつぎのようなものである．

1. そのブロック内で後に続く演算を検査して，A への代入が生じていれば，上の定義は死んでいる．A を使用するものがあれば，A は死んではいない．もし，A を参照するものがなければ，ステップ 2. へ進む．
2. このプログラムの全てのブロックに対する REF，ASS，RBA ベクトルが利用できるものと仮定して，この代入が生じたブロックからでている全ての道をたどる．もしあるブロックの RBA が A の代入の前にこれを使用することを示していれば，A は死んではおらず，処理はここで終る．もし ASS が，何かの代入を示していれば，この道はこれ以上たどる必要はない．もし全ての道がこのまま終るか，ループを始めたら，この変数は死んでいる．

18.4 議論および参考文献概説

畳み込み，冗長演算の除去

算術式を，特定の機械語に翻訳しながら同時に限定された最適化を行うことを論じた初期の論文がいくつかある．これらの中に Knuth (59) と Huskey

(61) がある．Floyd (61 a) のアルゴリズムは store と load の演算の回数を減らし，畳み込みを行い，冗長演算の除去を実施するものである．これはまた負号を含んだ式の等価性も識別している（例．$-b+a=-(b-a)$）．Gear (65) もこの技法を議論している．Hopgood (69) は，内部形式に木構造を用いて一連の代入文を限られた形で最適化する方法を議論している．

Breuer (69) は，一連の式と代入文から共通演算を括り出す方法を述べ，より多くの共通式が検出できるようになっている．したがって

```
J := A + B + C + E; K := A + B + E + F;
L := A + B + D + E + F;
```

は

```
T := A + B + E; J := T + C; K := T + F; L := K + D
```

と変換され，

(18.4.1)　　`A * B * C + A * B * D + A * E + F` から
　　　　　　`A * (B * (C + D) + E) + F` が得られる

一たん共通因子が決まり変換が行われると，load と store の演算回数を減らすために演算を並べ直す（これは後述する）．

共通因子を見出す手続きは，発見的なもので最適な共通因子を見出し得ないこともある．この手続きを実用的なものにするために，因子になりうることを試してみる相異なる要素の数を制限することである．

Fateman (69) が述べているように，共通因子の括り出しによって必ずしも最適なコードが得られるわけではない．例えば Breuer の方法では $5+x+x^2+7x^4$ は $5+x*(1+x*(1+x*(0+7*x)))$ と変形されるが，つぎのように変形した方がさらに加算と乗算を1個づつ減らすことができる：

$$T1 := x*x;\ T2 := T1*T1;$$
$$RESULT := 5+x+T1+7*T2;$$

Fateman によるもう一つ重要な指摘は，逐次的計算に対する最良コードが必ずしも並列的計算に対する最良コードではない点である．例えば，(18.4.1) の下の式の評価には5ステップ要するのに，独立な算術をいくつか同時に評価できる計算機では，この式は4ステップで評価することができる．これはつぎのようにする．ただし各行が1ステップに対応する．

```
T1 := A*B          T2 := A*E
T3 := T1*C         T4 := T1*D
T3 := T3 + T4      T2 := T2 + F
RESULT := T3 + T2
```

全ての言語で式の変形ができるわけではないことも知っておかねばならない．さらに，式の変形を行うことによって浮動小数点数を扱う演算では結果が異なる可能性がある（異なるのが普通である）．これらのことを要約すると代入文や式をいつどのように最適化するかについてはほとんどわずかなことしかわかっていない．

ループの最適化

Gear (65) は，18.2 節で述べたループの中程度の最適化——不変演算の処理とループの制御変数を含む演算の強さの軽減——を議論している．彼は n 項演算子 + および * の被演算子の並べ換えについても論じている．

Busam と Englund (69) も同じ目的で，適度の最適化とコンパイル時間の高速化とを調和させる問題を扱っている．彼らは不変式をループの外に移動させているが，演算強さの軽減は添字式についてのみ行っている（下記を見よ）．すなわち，彼らの用いた方法の特徴で興味深いものは，変数（あるいは式）がループの中で代入されるか参照されるかを決定するものである．そのような要素の各々に対して，代入される各点をつなぐ1次元リストと，参照される各点をつなぐ1次元リストを用意し，これらのリストは，内部形式プログラムの中にそのままおいておく．これは 13.3 節で述べた，未定義の名札の参照を鎖でつないだと同じ方法である．最適化処理が進むにつれて，ある変数の記号表のエントリには，最後に処理した代入点 ASS と参照点 REF を示す情報が入

```
   :
10 I {50} =
20 DO 40 J = 1, N
   :
   K = I * 5
   :
40 CONTINUE
50 I {0} =
```

記号表でIのエントリの定義の鎖
欄には10が入っている

図18.14 定義の鎖

る．したがって，**図18.14** で定義の鎖（代入点の1次元リスト）から演算 I＊5 がループ内で不変であることを容易に知ることができる（図中の中かっこは内部形式の被演算子の1次元リストの連結欄を示している）．

ALGOL コンパイラの中には，Busam と Englund が行ったように，添字付変数の番地計算をループの外に移動する機構を備えたものがいくつか実現されている．これについては細かく議論しなかった．というのは 18.2 節に述べた最適化法が自動的にこれを含んでいるからで，このためにコンパイル時間が増えることはないか増えてもわずかである．これについては，Hill ほか (62)，Huxtable (64)，Gries ほか (65) を参照されたい．再帰的番地計算法と称されるこの方法は Samelson と Bauer (60) が最初に論じている．

IBM の FORTRAN H コンパイラ (Lowry と Medlock (69) 参照) と Allen (69) が実験的に設計したコンパイラでは，18.3 節に述べた，より高度な最適化の大部分を採用している．内部形式プログラムとしては Lowry と Medlock は 4-組を，Allen は間接 3-組を使用している．ともに基本ブロックを用いてプログラムを形成する．Allen の論文には実現法がより詳しく述べられている．Cocke と Schwartz (70) にも，これらの技法の議論などが含まれている．

レジスタの割付

これまではでき上った目的プログラムをさらに最適化することは論じなかった．各計算機ごとに異なるからである．これについてここで簡単に触れよう．

高速レジスタが複数個あり，それに中間結果を置くことができるような計算機を仮定すると，最初に思いつく最適化法の一つは，必要なレジスタの数を最小にするような算術式の評価方法を決定することである．Anderson (64)，Nakata (67)，Redziejowski (69) がこの問題を扱っている．論旨はつぎのとおり：A＊B のような式の評価にあたって，もし A の評価に n 個のレジスタを要し，B の評価に m 個のレジスタを要するならば $(m>n)$，A＊B の評価で，A を先に評価すると $m+1$ 個のレジスタが必要となるのに（A の結果を保持するために1個余分に必要）B を先に評価すれば，m 個でよい．

式は計算機内部で木構造で表現される（11.4 節参照）．各被演算子に必要なレジスタの個数を各演算子ごとに決定するために，この木を走査し各演算子の節点ごとにどちらの被演算子を先に評価するかを示すしるしをつけるパスを追加する．この後，そのしるしにしたがってコードを生成する．

Nakata もレジスタの数が十分にないときにどのレジスタを退避すべきかを決定するための 発見的規則をいくつか示している．例えば，A − B や A/B のような演算に対して，大抵の計算機では，A がまず累算器の上になければならないので，レジスタが必要だからといって，このような A の値を先に退避すべきではない．Redziejowski はこの問題をグラフ理論を用いて定式化し，レジスタ個数の最小化問題の特別な場合に適用できる一般的なアルゴリズムを展開している．

上述の方法の主な欠点は，レジスタの数が不足したときどのレジスタを退避すべきかを示さないところにある．n 個の高速レジスタが与えられたとき，ある式を評価するために必要な load 命令と store 命令の個数を最小にすること，この問題こそ実際に解決したいものである．この議論を始めるために，部分式を並べ換えることは認めず，その式の値は使用される前に高速レジスタの上にのせられていなければならないと仮定しよう．

n 個のレジスタを $R_1, R_2, \cdots, R_n$ とする．式が評価されるにつれて，ある時点で変数 V の値が必要となる．この時，つぎのような可能性が生ずる：

1. その値はすでにレジスタ R_i にのっている．このレジスタは使用中．
2. V にはまだどのレジスタも割当てられていないが，空のレジスタが使用可能である（あるいは，もはや内容が必要でないレジスタがある）：V をそのレジスタにロードする．
3. V に割りつけてあるレジスタはなく，全てのレジスタが現在使用中である．あるレジスタの内容を退避し（これは後に再びロードされる），V をこのレジスタにロードする．

問題は，この第三の場合にどのレジスタを退避すべきかである．直観的な答は

(18.4.2) 演算の系列の中で最も遠くで使われているレジスタを退避する

というものである．このレジスタ割付法は，かなりよく知られており，いくつかのコンパイラで用いられてきた．Belady は，ある条件のもとで，このレジスタ割付法は最適である——これによって load と store 命令の個数を最小にできる——ことを証明した．

このアルゴリズムを用いると各中間結果用変数がつぎにどこで用いられるかきちんとわかっていなければならないことに注意．これを知るには，目的コードの中に，参照の1次元リストを通しておくのが最善であろう．

Horwitz ほか (66) も，この問題に取組んだ．彼らは，基本とする規則を多少変えている．彼らは，演算の系列は順序どおりに実行されるものとして，一つの演算は，一つの値を参照でき（したがってその値は高速レジスタにロードされねばならない)，一つのレジスタの内容を変更できると仮定する．解くべき問題は，なお load 命令と store 命令の個数を最小にするものである．彼らのアルゴリズムにはグラフ上である節点から他の節点への最短経路を見出すものが含まれている．Luccio (67) は，彼らの結果を，多少拡張して，グラフの大きさを小さくしそれによってこのアルゴリズムを一層実用的にすることを目的としている．

Sethi と Ullman (70) も load 命令と store 命令の個数を最小にするアルゴリズムを示している．基本となる規則も多少異なる．すなわち，彼らは + と * を用いている算術式と，被演算子が全て単純変数で相異なるものを考えている（したがって共通な部分式はない)．演算は並べ換えが可能である．

並列演算用の最適化

CDC 6600 計算機は，複数個の独立な演算装置を持ち，それらは全て並列に実行することができる．問題は，この並列性の効果を発揮できるコードを生成するものである．Allard と Wolf と Zemlin (64) は，算術式に対するこの種の最適化の試みを述べている．主な方法は，演算を並べ換えて，冗長な load 命令や store 命令を除去し，レジスタの割付を行うものである．Hellerman (66) と Squire (63) も，並列演算装置に対して算術式を複数パスでコンパイルする方法を報告している．Stone (67) は，1 パスアルゴリズムを与えている．例えば A + B + C + D + E + F + G + H を内部形式に変換する際に

((((((A + B) + C) + D) + E) + F) + G) + H

とする代りに

((A + B) + (C + D)) + ((E + F) + (G + H))

と 1 パスで変換する．これによって A + B, C + D, E + F, G + H が並列に計算できる．Baer と Bovet (68) も参照されたい．

その他の最適化

McKeeman (65) は“のぞき穴 (peephole)”の最適化を述べている．これは，生成された後の目的プログラムを処理するもので，一時に眺めるのはわずか数命令である．これによって，短かい命令系列に注目してそれをもっと能率のよい命令系列に置換えることができるようになる．例えば，IBM 7090 で

```
CLA  A        は        CLS  A
CHS
```

に変換することができる．このような方法で，短かい命令系列の内部で冗長な load 命令を見出すこともできる．

Yershov (66) は ALPHA——ALGOL を拡張したもの——に対して，ソ連製の計算機 M-20（1 語 45 ビット，3 アドレス命令，4096 語）に書いた 24-パスコンパイラを述べている．通常の for ループの最適化と冗長な演算の除去のほかに，つぎのようなことを実施している．

1. 隣合った for ループを可能ならば結合して単一の for 節にする（ループ融合）．
2. 同一の記憶場所を共有できる変数を探して変数に割付ける語数を最小にする．
3. 一度しか呼ばれない手続きがあれば，最適化とコード生成を行う前に，これを原始テキストの段階で置換える．
4. ある手続きの特定の仮引数に対する実引数が，この手続きを呼出す全ての場合に同じであれば，コンパイル時に実引数を仮引数に置換えておく．

Wagner は学位論文で，被演算子が配列であるような式について，中間結果用配列の記憶場所を少くするための最適化を述べている．彼は上述のループの融合を詳しく論じている．Busam と Englund は入れ子になったループをいつどのようにして単一のループにするかを述べている．

このほか，つぎのような問題がある：

1. 除算 A/B を A*(1/B) に変え，1/B をループの外に出し，内側でより高速な乗算を実現する試み．
2. 整数型定数を指数とする演算を乗算で実行すること．例えば，X**2 は X*X，Y**7 は (Y*Y)*(Y*Y)*(Y*Y)*Y など．
3. 2進法の計算機では，2を底とする整数の乗除算は，桁移動で実行でき

る．2.0 を乗ずることは加算に直せる．

4. ABS, ENTIER ほか数学的関数をそのままコードの系列の中に埋め込む．

5. 一つづきの記憶場所の移動は1命令に圧縮する．

6. 基本ブロックに対するコードの生成された順序を変更して不要な無条件分岐命令を除去する．

7. 1 を掛けたり 0 を加えたりする演算の削除

8. 実行されるはずのないコードの削除（これは場所の節約で時間とは無関係）．

第19章

マクロの使用

本章では，高水準言語にマクロの機能を盛込む可能性を探り，そのような機能を実現する方法を示したい．19.1節では簡単なマクロ機構を述べ，それを計算機で実現する方法を詳しく示す．19.2節では，さらに能力と融通性を持ったマクロ機構へ拡張することを述べる．この拡張の実現法については簡単に述べる．19.3節では，よく知られている独立型のマクロ処理系について述べる．それは特定の言語と密接に関係しているものではなくて，多くのプログラム言語のコンパイラの前処理用として使用しうるものである．最後に19.4節はマクロ処理系の種々の面に関する文献に言及する．

19.1 簡単なマクロ機構

マクロの基本的な考え方は，テキストの代入，あるいは挿入である．すなわち，原始プログラム中のある名前が別の文字列で置換えられることである．これを図19.1で説明しよう．まず，**マクロ定義**で，MACという名前と，文字列"ST VW"とを関連づけておくとしよう（図19.1a）．原始プログラムを走査して，MACという名前に出会うたびに，これを文字列"ST VW"と置換える．そこで，プログラマが自分のプログラムを図19.1bのように書くと，実際にコンパイラに渡される原始プログラムは図19.1cのようになる．

マクロ定義名に関連づけられた文字列を**マクロの本体**と呼ぶ．MACという名前は，それが原始プログラムに現れる時に，**マクロ呼出し**となる．マクロ呼出しにマクロの本体を代入する過程をマクロ展開と呼ぶのが一般的である．

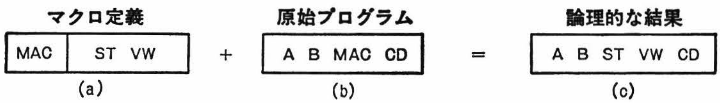


図 19.1　簡単なマクロの挿入

さらに融通性を持たせるために，マクロに引数を許すことがある．マクロ呼出しのたびに引数によって，それと置換える文字列が変化する．図19.2はこれを説明している．マクロ F に関連づけられている文字列はいわゆる仮引数を2個——&1 と &2——持っている（図19.2a）．これに対応して，F のマクロ呼出しには2個の実引数——XY と XZ——がある（図19.2b）．このマクロ呼出しを本体と置換える時に，XY は第1仮引数&1，XZ は第2仮引数 &2 とそれぞれ置換わって，結局図19.2c となる．

ここで“仮引数”とか“実引数”という言葉を用いたが，これは普通手続きやサブルーチンで使うものである．つまりマクロ呼出しが手続き呼出しと大変よく似ているからである．しかし大きな違いが一つある：それはマクロ呼出しがあるとテキストの代入が生じてその後にコンパイラがそのプログラムを構文解析するという点である．これは“マクロ処理系”がコンパイラとは完全に分離されている場合で，この出力がコンパイラに渡される．ここではコンパイラの走査部の中でマクロが実際どのように処理されてゆくかを示そう．

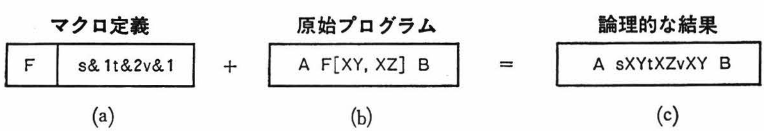


図 19.2　引数を持った呼出しに対するマクロ展開

入れ子になったマクロ呼出しと定義

マクロ呼出しが，マクロの本体の中や実引数の中に現れることがしばしばある．図19.3は PI，ADD という2個のマクロ名を用いてこれを説明している．マクロ呼出し PI は，マクロ ADD の本体の中（図19.3 a）とマクロ呼出し ADD の2番目の実引数（図19.3 b）に現れている．三つのマクロ呼出しを全

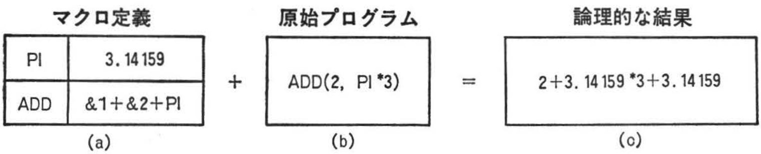


図 19.3 入れ子になったマクロ呼出し

て展開した結果が図19.3cである.

PI のようなマクロは，かなりしばしば現れる．特定の定数に記憶しやすい名前を割当てるとプログラムが読易くなるものである．さらに，この定数を何かの理由で変更しなければならないときにはそれを1ケ所で——すなわちマクロ定義の中で——変更すればよい.

マクロ呼出しでは，**入れ子の水準**の概念を持出すと便利である．すなわち，マクロ展開が行われないときは，水準0にあるという．水準nのマクロ展開のさなかに出会ったマクロ呼出しは，水準$n+1$を持つ．図19.3の例では，マクロ呼出し ADD は水準1，マクロ呼出し PI は二つとも水準2である.

入れ子になったマクロ定義は，それほどしばしば現れるわけではないが，時には役に立つことがある．図19.4はマクロ A の定義でその本体にマクロ B の定義を含んでいる．(われわれはまだマクロ定義の正確な形式は定めていなかった．図19.4では MACRO B[XYZ] という定義はマクロ名 B の本体が XYZ であると定義している.) 入れ子になったマクロ定義の効果は，手続きに局所的な変数の効果によく似ている．マクロ A の外側では，B の定義は無効である．B のマクロ名と本体は，“マクロ表”の中にも出てこない．ただし，A のマクロ展開が行われている間はその限りではない．A の展開が終ると，B はマクロ表から削除される.

実引数の中に現れるマクロ定義も本質的には同じ方法で処理される．この定義はその引数が評価され，原始プログラムの中に挿入されている間だけ有効である.

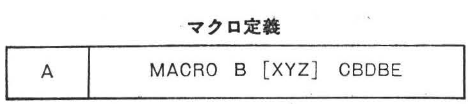


図 19.4 入れ子になったマクロ定義

マクロの定義と呼出しは原始プログラムのどこに現れてもよいことに注意しよう．これは，原始プログラムのブロック構造や文の構造には全く無関係である．なんとなれば，マクロは，プログラムがコンパイルされる以前に処理されるからである．マクロ処理の結果が構文上も意味上も正しい原始プログラムであるということがただ一つの制限であることはいうでもない．

マクロの定義と呼出しの形式

マクロの定義と呼出しには，特別な記号&,[,および]を用いる．これらは原始プログラムの記号ではないと仮定する．この制限によってマクロ展開の過程とコンパイルの過程をはっきりと分割することができるようになる．記号“,”もマクロ呼出しで使用されるが，これは原始プログラムの記号である場合が多い．

マクロ定義は，つぎの形式による：

(19.1.1)　MACRO 〈マクロ名〉[〈文字列〉]

ここで〈マクロ名〉は，プログラム言語における通常の〈名前〉であり，〈文字列〉は，文字の任意の系列である．二つのかっこの間の文字は空白も含めて，全て〈文字列〉の要素である．さらに〈文字列〉中では，角かっこは対になって現れるという制限をつける：すなわち左かっこ[に対してそれより後の適当なところに対応する右かっこ]が来なければならない．最後に〈文字列〉中，文字&が現れるとその次の文字は必ず0以外の数字でなければならない．&i をマクロの i 番目の仮引数と呼ぶ．この使用法は，直ぐ説明しよう．マクロ定義というのは，〈マクロ名〉とその本体——すなわち〈文字列〉——とを結び付けることである．マクロ定義を走査すると，マクロ名とその本体はマクロ表に格納され，その定義は原始プログラムから削除してしまう．

マクロ呼出しはつぎのいずれかの形式である：

(19.1.2)　〈マクロ名〉

(19.1.3)　〈マクロ名〉[〈引数〉,〈引数〉,···,〈引数〉]

ここで〈マクロ名〉はすでにマクロ定義で定ったものである．実引数〈引数〉は，つぎのいずれかの形である：

(19.1.4) [〈文字列〉]

(19.1.5) コンマも角かっこも含まない任意の文字列

もし〈引数〉が (19.1.4) の形式であれば，実引数は〈文字列〉であり，これは前に〈文字列〉に与えた仕様と一致したものでなければならない．第二の場合 (19.1.5) は実引数は空白以外の最初の文字に始り，コンマあるいは右かっこ] の前で終る．したがって，意味のある空白を前に付けたければ (19.1.4) の形式を採らねばならない．ほかにもう一つ制限をつける．これには二つ理由がある．一つはこれから示すマクロの計算機への実現を簡単にし状況がたやすく把握できるようにしたいからである．第二は，読者に自分で実現してみる機会を持ってほしいからである．演習問題1と2ではこの制限を取除いてある．

制限というのは，実引数には仮引数 &i を含んではならないということで，これはマクロ定義の中でマクロ呼出しを使おうとすると納得できないかも知れない．例えば，B がすでにマクロとして定義されているとして

```
MACRO A [ B [c &1 d] ]
```

の中で &1 はマクロ B ではなくマクロ A の呼出しの第1実引数を参照すべきであるが，われわれの制限はこのような使い方を禁止している．

マクロ呼出しはつぎのように展開される：

1. 実引数を走査して別の場所に転写し，マクロ呼出しを原始プログラムから削除する．
2. マクロ表（マクロ名とそれに対する文字列が入っている）を探してこの〈マクロ名〉に対する最新の定義を見出す．
3. マクロの本体の文字列を原始プログラムに一つづつ挿入する．仮引数 &i に出会ったらマクロの本体からの記号を挿入するのを中断し，i 番目の実引数（必ず存在しなければならない）からの記号の挿入を始める．その引数が終ると実引数の処理中に追加されたマクロ定義はどれも削除し，マクロの本体の処理に戻る．
4. ステップ 3. の進行中に新しいマクロ定義やマクロ呼出しに出会うことがある．このときは，直ちにそちらの処理を行う．したがって，マクロ呼出しは再帰的である．
5. ステップ 3. が終るとこのマクロ呼出し中にマクロ表に載せられたマクロはどれも表から削除される．

簡単なマクロ機構の実現

以下に示すプログラムで，誤りの検査や，あふれの検査の多くには触れずにわかり易い説明に努める．

ここで，マクロ定義をおさめるために三つの配列を使用する．マクロの本体は，配列Sの1要素に1文字づつしまい，本体の最後の文字の次に#を付加える．原始プログラムには#は現れないと仮定している．配列MACNAMEには，定義済みのマクロ名が入り，配列MACBODYには配列Sにあるマクロ名に対応する本体への指標が入っている．配列CALLSには，現在展開中のマクロ呼出し（複数ありうる）に関する情報が入っている．図19.5は必要な変数の（完全な）一覧表である．図19.6は図19.3のマクロPIおよびADDの配列の有様を示した．

マクロ処理の実現は全て走査部の中で実行される．そこで走査部の中でどんなことが行われているか簡単に説明しよう．まず第一に普通の原始プログラムが走査されているときは大域的な変数INPUTに0が入っている．しかしマクロの本体や実引数が原始プログラムに挿入されている間は，われわれの持っている入力の流れは別のところから来ている．このような場合INPUTには配列Sに入ったマクロの本体や実引数の次に走査すべき文字へのポインタが入って

STRING ARRAY MACNAME(1:100);	マクロ名を入れる.
INTEGER ARRAY MACBODY(1:100);	マクロ名に対応する本体(配列S中にある)への指標を入れる.
INTEGER NMAC;	MACNAME中のマクロ名の個数.
CHARACTER ARRAY S(1:1000);	マクロの本体，マクロ呼出しの引数を1要素に1文字で入れる.
INTEGER NS;	S中の文字数.
INTEGER ARRAY CALLS(1:50);	現在のマクロ呼出しに関する情報を入れる(スタック風に働く).
INTEGER N;	CALLS中の要素の数.
INTEGER INPUT;	もし0なら入力はカードから,0でなければ,S中の現在の入力流の文字への指標が入る.
INTEGER LBSIGN;	原始プログラムを走査するかマクロ定義や実引数を移しているとき値が0.
CHARACTER CHAR;	現在入力された文字.
STRING ID;	最新の名前.

NMAC, NS, N, INPUT, LBSIGNの初期値は全て0.

図19.5　マクロ処理に必要な宣言

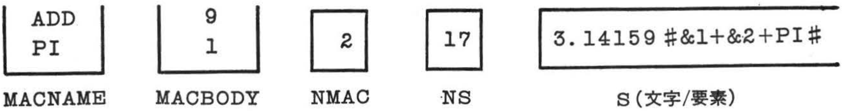


図 19.6 マクロ PI と ADD に対する各表の内容

いる．変数CHARには入力がカードであれ，マクロ本体であれ次の入力文字が入る．手続きGETCHARは走査部のあらゆる部分で使用され，つぎの入力文字を走査し，それを CHAR に代入するのがその機能である．INPUT の内容によってつぎの文字が原始プログラムから来たりマクロの本体あるいは実引数から来たりする．後の二つの場合には＃記号に出会うとそれがマクロの本体あるいは実引数を完全に走査し終ったことを示している．すると GETCHAR は手続き SWITCHBACK を呼出す．この手続きは入力流を以前のものに切換えて走査部がそこからの文字の処理ができるようにする．この手続きについては本節の最後で言及する．実際，実引数やマクロの本体を走査しているのでなければ＃が現れることがあってはならない．もし現れたとするとエラーメッセージを出す．変数 LBSIGN の値で＃が現れうるか (1)，否か (0) を示している．GETCHAR の大略の流れを図 19.7 に示した．

図 19.7 から分るように手続き GETCHAR は文字＆を見ると入力流を新たに実引数の方に切換えるために FORMPAR ルーチンを呼出す．

さらに走査部手続きが二つ必要である．一つは GETNONBLANK でこれは CHAR が空白かどうかを検査する．もし空白ならば空白以外の文字が現れるま

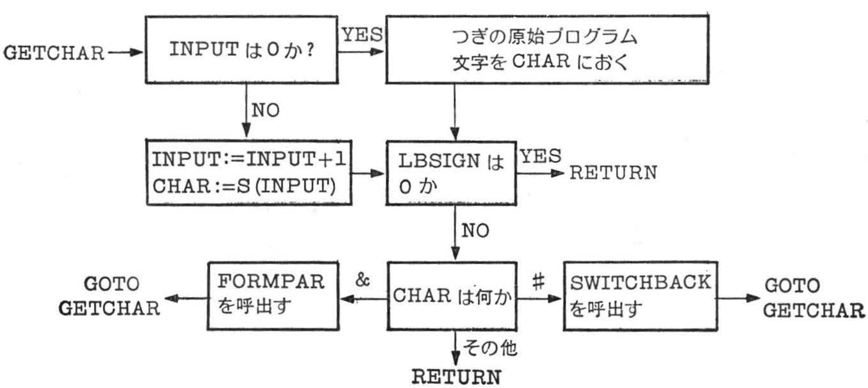


図 19.7 手続き GETCHAR

で繰返し GETCHAR を呼出す．第二の手続きは BUILDID で予約語や予約名の先頭の文字を走査したときに呼出される．BUILDID は GETCHAR を使ってあとの文字を走査し，名前を作り上げ，それを文字型変数 ID に代入しつぎの文字を走査して戻る．

これでマクロを実現させる手続きを書く用意が整った．手続き MACRODEF は走査部が予約語 MACRO を認識すると呼出される．MACRODEF はマクロ名を走査しそれとSの中へのポインタ（そこから本体が格納される）をそれぞれ二つの配列 MACNAME と MACBODY にしまい，手続き GETSTRING を呼んで，実際に本体を配列Sに移す．ここで特別な手続きが使用される．というのも，実引数の処理に対しても本質的には同じことをしなければならないからである．この手続きは "#" に出会うはずがないので変数 LBSIGN に 0 をセットする．もし#に出会うようなことになればエラーメッセージを出すことに注意．マクロ定義は全て同じ入力流から来るべきものである．

PROCEDURE MACRODEF

INTEGER LB; LB:=LBSIGN; LBSIGN=0;	旧 LBSIGN の値を退避し0に設定して#には出会わないことを示す.
GETNONBLANK; BUILDID;	マクロ名を取ってくる.
NMAC:=NMAC+1; MACNAME(NMAC):=ID; MACBODY(NMAC):=NS+1;	マクロ名とSへの指標（その本体が入る場所）をそれぞれ表にのせる.
GETNONBLANK; IF CHAR ≠ "[" THEN ERROR; GETSTRING;	マクロ名の直後に [が来ていることを確かめSの中にマクロの本体を移すためのルーチンを呼出す.
LBSIGN:=LB; GETCHAR;	#に関するシグナルを消してつぎの文字の走査に移る.

手続き GETSTRING は現在の入力文字列から1文字づつ配列 S に移すルーチンである．この手続きに入るに際して CHAR には左角かっこがあり，その後に文字列が続く．この文字列上の最初に続く空白もそのまま転写されることに注意．GETSTRING はかっこの対を数えていてどこで止まるかを見張っておかねばならない．このために局所的な変数Cを用いる．

PROCEDURE GETSTRING

INTEGER C; C:=1;	かっこ対カウンタを1に設定.
WHILE C ≠ 0 DO BEGIN GETCHAR; NS:=NS+1; S(NS):=CHAR; IF CHAR="[" THEN C:=C+1; IF CHAR="]" THEN C:=C−1; END;	かっこ対カウンタの値が0になるまで文字を走査しそれを配列Sに移し続ける.
S(NS):="#"	終ったときには最後の] を#に書変える.

つぎのステップはマクロ呼出しの開始方法を説明することである．必要な仕事は，現在の入力流やマクロ表や配列Sに関する情報を退避することである．この情報はマクロ呼出しが終った所でもとに戻される．マクロ呼出しは他の呼出しの展開を行っている最中にも起りうるので上述の情報を保持するためには，スタックを使用する．マクロ呼出しのたびに CALLS スタックの要素を数個使用する（CALLS は図19.5では配列として宣言）．それぞれの呼出しに対しては，図19.8 **a** のような形式になっている．

走査部が名前を識別するとそれをMACNAME表の最新の要素から始めて最初のものへと下向きに探す．見付かれば，MACNAME 表のエントリ番号が手続き MACROCALL の実引数として渡される．MACROCALL は分岐や条件のない真直ぐな手続きである．CALLS スタックの取扱いを別にすればこの手続きは実引数を S に移しマクロの本体の走査を始める．LBSIGN は再びゼロにセットされる．何となれば，マクロ呼出しは全て同じ入力流の中に現れなければなら

(a) マクロ呼出しの場合
旧 LBSIGN
旧 INPUT
旧 NMAC
旧 NS
k（引数の個数）
第k引数のS中への指標
:
第2引数のS中への指標
第1引数のS中への指標

(b) 引数呼出しの場合
旧 LBSIGN
旧 INPUT
旧 NMAC
旧 NS
−1

図 19.8 CALLS スタックの様相

PROCEDURE MACROCALL(I); INTEGER I;

INTEGER LB, TEMP;	Sの現在の最上段へのポインタを退避しあとで回復できるようにしておく.
TEMP:=NS;	
LB:=LBSIGN; LBSIGN:=0;	呼出しの中で#に出会ってはいけない.
N:=N+1; CALLS(N):=0;	引数の数の初期値を0にし最初の空白でない文字を取出す.
GETNONBLANK;	
IF CHAR="[" THEN WHILE CHAR ≠"]" DO GETPARAM;	呼出しに実引数があればGETPARAMを呼んで実引数を全てSに移す.
N:=N+1; CALLS(N):=TEMP; N:=N+1; CALLS(N):=NMAC;	SおよびMACDEFの現在の最上段へのポインタをCALLSスタックにのせる(呼出しが終るとこれらをもとに戻す)
N:=N+1; CALLS(N):=INPUT; N:=N+1; CALLS(N):=LB;	現在の入力流へのポインタ,停止条件(#)スイッチをスタックにのせる.
LBSIGN:=1;	マクロの本体の走査を始めるのだから最後は#まで.
INPUT:=MACBODY(I); CHAR:=S(INPUT);	マクロの本体へのポインタを設定し最初の文字を取ってくる.

ないからである. MACROCALL(I) で現在呼ばれているマクロが MACNAME 表の I 番目のものであることを示している.

つぎのルーチンは手続き MACROCALL から呼出されるもので一つの引数を現在の入力流から配列Sに移すものである. このルーチンはもし引数が"["で始まっていれば GETSTRING ルーチンを呼出し, さもなければ引数をそのまま引渡す. GETPAPAM は MACROCALL の理解を容易にするために作ったものでこの手続き自体は1ケ所でしか呼出されていないので実際には手続きにすることはない.

PROCEDURE GETPARAM

CALLS(N):=CALLS(N)+1;	引数の個数を1あげる.
N:=N+1; CALLS(N):=CALLS(N−1);	スタックの最上段を一つずらしてそこに
CALLS(N−1):=NS+1;	引数の番地を挿入し最初の文字を走査す
GETNONBLANK;	る.
IF CHAR="[" THEN	もし引数が [で始っていれば, 〈文字列〉
BEGIN GETSTRING; GETCHAR;	を移すルーチンを呼出しつぎの空白でな
GETNONBLANK;	い文字を取出す.
IF CHAR ≠ "," OR CHAR ≠ "]"	その文字はコンマか] でなければならな
THEN ERROR	い.
END	ここではコンマやかっこを含まない実引
ELSE	数の引渡しを行う.
BEGIN	
WHILE CHAR ≠ "," AND CHAR ≠ "]"	実引数の終りの区切りが検出されるまで
DO BEGIN NS:=NS+1;	1文字ずつ S に移す.
S(NS):=CHAR; GETCHAR;	
END;	
NS:=NS+1; S(NS):=#	最後に終りの印#を書込む.
END	

PROCEDURE FORMPAR

INTEGER TEMP; GETCHAR††;	CHAR に引数の番号を取出す.
IF N=0 OR CALLS(N−4)<0 OR	マクロの展開をしなければならないのはマク
CHAR は数字ではない OR	ロであって実引数ではない. CHAR はマクロ
CHAR="0" OR	呼出し中にある実引数を参照する整数値でな
CHAR† >CALLS(N−4)	ければならない.
THEN ERROR;	
TEMP:=CALLS(N−4);	CHAR によって決まる番号の引数の在場所
TEMP:=N−5−TEMP+CHAR†;	を示す要素の CALLS 中の指標を TEMP
N:=N+1; CALLS(N):=−1;	に.
N:=N+1; CALLS(N):=NS;	ついで実引数の呼出しに必要な要素を5個
N:=N+1; CALLS(N):=NMAC;	CALLS スタックに置く (図19.8(b) 参照)
N:=N+1; CALLS(N):=INPUT;	
N:=N+1; CALLS(N):=LBSIGN;	
LBSIGN:=1;	実引数の走査は#で終ることを示す.
INPUT:=CALLS(TEMP)−1;	入力流を実引数に切換えて戻る.

FORMPAR ルーチンは, GETCHAR から呼出されて仮引数 &*i* を処理する.†† 仮引数は引数を持たないマクロ呼出しと云ってもよいのでここではこれをそのように取扱っていることに注意しよう. 図 19.8(**b**) のような形式の要素を CALLS

† 訳注　CHAR に入っているのは 1～9 の文字であるから, 実際にはこの文字を対応する整数値に変換してから演算しなければいけない.

†† 訳注　FORMPAR は再び GETCHAR を呼出している.

スタックに加える．FORMPAR に入る際には&が走査されたばかりである．ただ一つの違いは，仮引数は実引数の中に現れることがない点である．したがって，CALLS スタックの第1要素として "−1" を使うのは，それを実際のマクロ呼出しと区別するためである．FORMPAR は & のつぎに続く数字を見て二，三誤りの検査を行う．そして CALLS スタックに値を置いて入力流を実引数の方に切換えその最初の文字を CHAR に置く．

最後はマクロの本体や実引数の走査の終了をプログラムする仕事である．両者とも似たようなものなので同じルーチンによって取扱う．GETCHAR が # を認識すると（この # は S 中のそれぞれのマクロ本体の終りと実引数の最後に置かれたもの)，手続き SWITCHBACK を呼出す．SWITCHBACK は，スタックの最上段にある要素が明示する，現在の呼出しを開始する前に退避しておいた情報を回復する．

PROCEDURE SWITCHBACK

LBSIGN:=CALLS(N);	直前の終了信号を回復．
INPUT :=CALLS(N−1);	直前の入力流のポインタを回復．
NMAC :=CALLS(N−2);	現在の呼出し処理中に加わったマクロや実引数を正
NS :=CALLS(N−3);	しく取除くために NMAC と NS の値を回復する．
N:=N−5;	現在の呼出しで用いたスタックの要素を取除く．
IF CALLS(N+1)>0	もし CALLS(N+1) > 0 ならばさらに別の実引数
THEN N:=N−CALLS(N+1);	を取除かねばならない．

演習問題　19.1

1. 本節では実引数の中には仮引数 &i が現れないという制限を設けた．もしこの制限をはずすとすればそのような仮引数の用法を説明せよ．どの呼出しのどの実引数にその仮引数が対応するのか．
2. 上問で述べたように本節の制限を取除いた時に本節で述べたマクロの実現法を修正せよ．

19.2　種々のマクロ

前節で概説した特別のマクロ機構は文字単位のものだった．すなわち原始プログラムは連続した文字の系列として単純に取扱われた．アセンブリ言語や FORTRAN などを含めて多くの言語はカードの書式に依っている．すなわちカード1枚に1文（1命令）という形式である．あるアセンブリ言語のマクロ

定義はつぎのような形式になろう.

```
MACRO
PLUS  &1, &2, &3
FETCH &1
ADD   &2
STORE &3
END
```

ここでマクロ名は PLUS であり本体はカードの3枚目から5枚目, END はマクロ本体の終りを示している. マクロ呼出し

```
PLUS  X, Y, Z
```

によって

```
FETCH  X
ADD    Y
STORE  Z
```

が生成される. このようなマクロ機構は前節に述べたものより実現は容易であろう. なんとなればマクロ呼出しはたった1ヶ所, 命令部にしか現れないからである. しかし実現法の大略の考え方は同じである. マクロ定義は場所を別に設けてそこに置き, マクロ呼出しが生ずると実引数を仮引数に代入しながらマクロの本体と置換える.

マクロの内部形式

もしマクロが適当な方法で定義されると, マクロの本体は文字のままよりも内部形式にして格納する方が一層能率がよい. マクロ定義が最初に走査されると, 一度で内部表現に翻訳される. 以後のマクロ呼出しに対する展開の過程がずっと速く簡単になる. 上のマクロ定義PLUSに対する内部形式はつぎのようになろう.

マクロ本体の表

CARD	1	
STRING	5	
OPFIELD		
PARAMETER	1	
CARD	2	
STRING	3	
OPFIELD		
PARAMETER	2	
CARD	3	
STRING	5	
OPFIELD		
PARAMETER	3	
ENDBODY		

FETCHADDSTORE

ここでマクロ本体の表の最も左側の欄には“新しいカードの始まり”，“仮引数をその中に含まない文字列”，“被演算子欄の始まり”，“仮引数”と“マクロ呼出し”を意味する数字が入っている．第2欄には，カード番号，文字列の文字数，あるいは仮引数の番号が入っている．3番目の欄は，文字列の場合に用いられ，別の表に納められた文字列へのポインタが入る．

このような内部形式の走査は，ずっと容易になる．何となれば，空白がすでに取除かれており，全ての可能性の検査が終って分類済みとなっているからである．マクロ呼出しを展開する過程は解釈過程であって16章で議論したものと非常によく似ている．そこには内部形式（マクロの本体）があり，これを解釈しなければならないのである．

ここで，よく用いるマクロを手短かに議論しよう．これには“マクロ処理時”における変数や中間結果用変数などの使用を必要とする．これによって，マクロ展開の過程が第16章の解釈過程と一層類似するようになる．

内成記号

マクロ本体の中で名札を使用すると，つぎの例に見るような面倒が生ずる：

```
MACRO
GREATER    &1, &2, &3
FETCH      &1
SUBTRACT   &2
GOIFMINUS  Z
```

```
    &3
Z : END
```

このマクロの呼出しが複数個になると呼出しのたびに同じ名札Zが生成される．この問題を避けるために大抵のアセンブリ言語のマクロには，何らかの方法で内成変数を自動生成できるような機能がある．一つの方法は&&という記号を前に置いてシステムに一意的な変数を生成することである．システムはこの記号を認めると変数が他の全ての変数と異なっていることを確かめる．システムの中には特別なシステム変数，例えば &MACNUM のようなものを持っているものもある．この変数の実際の値はアセンブル処理中，マクロ呼出しの回数を文字の形で表したものである．これを使って変数と連接して新しい変数を作る．例えば，上のGREATERはつぎのようになる：

```
           MACRO
           GREATER    &1, &2, &3
           FETCH      &1
           SUBTRACT   &2
           GOIFMINUS  Z&MACNUM
           &3
Z&MACNUM : END
```

原始プログラムの中で最初の二つのマクロ呼出しが

```
GREATER  A, B, "STORE B"
GREATER  B, A, "STORE A"
```

ならば，これからつぎのようなテキストが生成される：

```
     FETCH       A
     SUBTRACT    B
     GOIFMINUS   Z1
     STORE       B
Z1 : FETCH       B
     SUBTRACT    A
     GOIFMINUS   Z2
     STORE       A
Z2 :
```

マクロ処理時変数

マクロ処理時中に何らかの計算をしてしまうというのは有用なことが多い．このために計算の結果をしまう変数を使う必要が生ずる．OS/360 アセンブリ言語では，MACRO の中につぎのような文を許している．

```
LCLA  &X
LCLC  &Y
```

これは&X, &Y という変数がマクロに対して局所的に使われ，それぞれ算術型，文字型の変数であると定義している．

```
&X  SETA  &X+1
```

という命令が走査されるたびに，&X という変数には1が加えられる．後程，PL/I でこの機能がどうなっているか見ることにしよう．

条件付マクロ

マクロ処理中にある文の出力を押えたり，実引数に応じて別の文を出力したりすることができると便利なことが多い．例えば

```
PLUS  A, B, C
```

というマクロ呼出しによって

```
FETCH  A
ADD    B
STORE  C
```

を生成する PLUS マクロを

```
PLUS  A, 1, A
```

と呼出したときに限って

```
ADD1  A
```

を生成するようにしたいとしよう．ここで ADD1 は番地部で示す記憶場所に 1 を加えるために使われる特別な命令である．ここで必要なのは，何らかの方法で前処理の制御を出力すべき命令の位置に動かすことである．システムの中には命令の直前に特別の印を置いて，これは，前処理部によって処理させること

を陽に示すものがある．例えば，上のPLUSマクロをつぎのように書いてみよう：

```
          MACRO                            (1)
          PLUS       &1, &2, &3            (2)
          LCLB       &X                    (3)
&X        SETB       (&1  NE  &3)          (4)
          %GOIF      &X, %Z                (5)
&X        SETB       (&2  NE  "1")         (6)
          %GOIF      &X, %Z                (7)
          ADD1       &1                    (8)
          %GO        %Y                    (9)
%Z        %NULL                            (10)
          FETCH      &1                    (11)
          ADD        &2                    (12)
          STORE      &3                    (13)
%Y        %NULL                            (14)
          END
```

マクロが呼出されるとマクロの本体を走査する．3行目でマクロ時変数&Xが論理型と宣言されている（LCLB は local Boolean の略）．4行目で第1実引数を示す文字列と第3実引数を示す文字列とを比較する．もし両者が等しくなければ，マクロ時変数&X に TRUE が代入される．等しければ FALSE が代入される．5行目で&X を調べ，それが TRUE であれば %Z に制御を移す．%Z では，NULL 命令が実行され（何もしない），その後に11～13行目が出力される．もし5行目で&X が FALSE であれば，6～9行目の解釈に進む．

PL/I のコンパイル時機能

PL/I にはいわゆる“コンパイル時機能”がある (IBM(c)参照)．その大部分はマクロ処理である．ここでこの機能を概観しよう．完全な細部までの説明をすることはここでは不可能である．

PL/I コンパイラでは，まず前処理部が原始プログラムを走査する．プログラムが前処理されるにつれて，一部ずつ出力用バッファに送られ，適宜コンパイラに渡される．マクロ処理がなければ，全ての原始プログラムはそのまま出

カバッファに転写されるにすぎない．

PL/I にはマクロ定義というものはない．しかし文の直前に%記号を置くことによってその文が実行時ではなく前処理時に実行されるべきことを指示する．もちろん%を前に置けるのはいくつかの文：DECLARE, DO, END, GOTO, IF, PROCEDURE, NULL (空文)，に限られている．%を付けた手続きは，マクロに最も近い．このような手続き名が認識されると，この手続きは前処理時に実行され，その結果でき上った文字列がこの手続き呼出しと置換る．

簡単な例で前処理の考え方の大様を示そう．

```
DECLARE I FIXED;
DO I = 1 TO 5;
A(I) = B(I) * C(I);
END;
```

これは普通の PL/I プログラムの一部である．これはそのままコンパイラに渡されコンパイルされるとループが生成される．一方

```
%DECLARE I FIXED;
%DO I = 1 TO 5;
A(I) = B(I) * C(I);
%END;
```

では全く違ったプログラムが生成される．第一に，I の宣言は%が付いているので，I は前処理時変数であり前処理部からの出力中には決して現れない．第二に，このループは前処理時に実行される．ループが実行されるごとに，A(I) = B(I) * C(I) を走査して，I をその現在の値で置換えながら出力する．したがって，コンパイラに渡される実際のプログラムは

```
A(1) = B(1) * C(1);
A(2) = B(2) * C(2);
A(3) = B(3) * C(3);
A(4) = B(4) * C(4);
A(5) = B(5) * C(5);
```

このように前処理部がループの展開を行う．この例では，I が一種のマクロ名であり，その本体は常にその現在の値であるということもできる．

マクロ定義の第二の型は，%を付けた PROCEDURE である．これは前処理

部がこの手続きの呼出しを見ると，直ちに実行すべきことを示している．この手続きの呼出しの結果は，文字の系列を生成し，これが呼出しと置換わる．これは便利なシステムとなる可能性がある．しかしながら，そのまま転写される普通の文（例えば上のA(I) = B(I) * C(I)）を前処理の手続き中に書くことは許されない．許されるのは前処理時に直ちに実行できるものだけである．したがって，出力になりうるものは，前処理によって完全に組立てられた文字列だけである．

この前処理部は原始プログラムを走査し，前処理に関係する文を内部形式に変換することである．普通の PL/I の文はそのままにしておく．つぎのフェーズは第1フェーズの出力を16章で述べたような方法で解読する．解読中に通常の PL/I 文を走査し前処理時変数がその中にあれば，その現在の値で置換えて出力する．

19.3 汎用マクロ生成子 GPM

GPM (General Purpose Macro Generator) は英国ケンブリッジ大学数学研究所で，ALGOL 型言語 CPL のコンパイラを書くのを助ける目的で作られたマクロシステムで Strachey (65) によって考案された．GPM は19.1節で述べたシステムよりずっと工夫をこらしたシステムである．ここでは GPM の実現法については述べないがその難点をいくつか示すことにする．GPM を作製することは，読者にとって興味深い演習問題と考える（演習問題1）．

マクロ呼出し

GPM のマクロ呼出しの形式はつぎのいずれかである：

#〈マクロ名〉

#〈マクロ名〉,〈引数〉,〈引数〉, …,〈引数〉

ここで〈引数〉は実引数である．図 19.3 の PI と ADD を呼出すには

#ADD, 2, #PI; * 3;

のように書くことになろう．19.1節のシステムのようにまずマクロ定義でマクロの本体と結び付けられる．このマクロ本体は，一般に &1, &2, …, &9 のような仮引数を含んでいて，マクロ展開をする時に実引数と置換えられる．さらに &0 は，マクロ名自身と置換えられる特別の仮引数である．マクロ名

PLUS の本体が

A:＝&1＋&2＋&0

としよう．すると #PLUS, 2*X, 5; とマクロ呼出しすると

A:＝2*X＋5＋PLUS

が生成される．任意の文字列を区切り記号〈と〉で囲むことによって文字列の評価を後回しにする効果がある．この文字列はそのまま転写される．評価する代りに区切り記号〈と〉が取除かれる．これによって，文字列の区切り記号の開きかっこと閉じかっこが対応しない場合を除いて任意の記号を出力流に含ませることが可能となっている．例えばマクロ呼出し

#PLUS, B(1〈,〉2), 5; の結果は A:＝B(1, 2)＋5＋PLUS

であるし

#PLUS, B(1, 2), 5; は A:＝B(1＋2)＋PLUS となる．

後者の場合，引数5は評価の際に用いられていない．

マクロ呼出しが完了すると，そこに現れる全ての実引数と呼出しの最中に評価された任意のマクロ定義が削除される．前に述べたマクロ機構にあったような局所的なマクロ定義がここにもある．

マクロ定義

マクロ定義の機能は，特別なシステムマクロ DEF を用いることによって実現される．DEF の呼出しには2個の実引数，マクロ名とその本体を用いなければならない．したがって，図19.3のマクロ PI と ADD の定義はつぎのようになる：

#DEF, PI,〈3. 14159〉; #DEF, ADD,〈&1＋&2＋#PI;〉;

ここでマクロ定義がまた別のマクロ呼出しになることを再度注意しよう．この事がいかにわれわれの役に立つかについては後で触れる．

マクロ展開

GPM は一見19.1節の簡単なマクロ機構の表記法を改めただけで同じようである．これはある程度まで本当である．すなわち，平均的なプログラマが普通

に使う単純なマクロ呼出しでは両方のシステムで同じ結果が得られる．しかしながら主な違いはまだ実際には言及していない部分（すなわちマクロがいかに展開され，実引数がどうなるか）にある．GPM に許されている一般性のゆえに，GPM は巧妙で強力なシステムになっている．それではマクロ展開のアルゴリズムをのぞいてみよう．このアルゴリズムは十分注意して読まないとほんのつまらないものに思えるかもしれない．アルゴリズムを追いながら誤りやすい特徴のいくつかを指摘しよう．

入力流は左から右に走査され，マクロ呼出しがあるまでそのまま出力流に転写される（マクロ定義というのは特別なマクロ DEF の呼出しであったことを想起せよ）．呼出しが評価されその結果が呼出しに代って出力流に移され再びもとの入力流の走査を始める．マクロ呼出しはつぎのように評価される：

1. マクロ名と実引数が出現順に評価される．結果は適当な場所にたくわえる（もしこの評価中に何らかのマクロ呼出しに出会うとその呼出しが直ちに評価される．マクロ呼出しはこのように再帰的である）．
2. 呼ばれたマクロ名を KEY にしてマクロ定義表を最も新しいエントリから最初のエントリ（これはシステムマクロ DEF である）に向けて索引する．もし表中に名前がなければエラーメッセージを印刷する．
3. ステップ 2. で見付けたマクロの本体を，もとの入力流と同じ方法で走査する．すなわち，そこにマクロ呼出しがあれば直ちに展開する．もし仮引数 $\&i\ (i=0, 1, 2, \cdots 9)$ に出会うとステップ 1. で評価した i 番目の実引数の結果をそのまま複写して仮引数と置換える．&0 はそのマクロ名と置換えることに注意．もし i が引数の個数より大であれば誤りである．実引数が 2 度評価されることはないことに注意．処理中の実引数の中にマクロ呼出しを含んでいなければ複写されるだけである．マクロ呼出しの結果がこの走査の出力である．
4. 展開の終ったマクロ呼出し，評価の済んだ実引数，それにステップ 1.～3. の中で出てきたマクロ定義を完全に取除く．
5. もとの入力流の走査をさきのマクロ呼出しの最後のセミコロンのつぎの文字から再開する．

システムマクロ DEF の呼出しに対しては，ステップ 3. の処理を変更する．これには走査すべき本体がない．その代り DEF の呼出しで最初に評価した 2 個の実引数がそれぞれマクロ名と本体とみなされマクロ定義表に書込まれる．

マクロ展開アルゴリズムに関する注意

まずマクロ定義とは，特別なマクロ DEF の呼出しにほかならないのでマクロ名は単純な名前である必要はなく前後に空白を伴った任意の文字列でよい．このシステムでは空白の扱いは他の文字と本質的に同等である．

第二に，マクロ名とその実引数はともにマクロの本体の走査を始める前に（さらにマクロ定義表の検索を始める前に）評価される．したがって，これらは ALGOL における値呼びの引数に対応する．一方19.1節のシステムの引数は名前呼びの引数に似ていた．引数の効果の違いを見るために，つぎのようなマクロ呼出しを考えよう：

```
#A, X, Y, #DEF, A, B;;
```

すなわち3個の実引数 X, Y, #DEF, A, B; を持ったマクロ A を呼出している．他に何もしないうちに実引数の評価を行うのであるから，まず X を評価し（X ができる），Y を評価し（Y ができる），そしてマクロ定義 #DEF, A, B; を評価する．この評価によってマクロ定義表に本体Bを持ったマクロAを付加えることになる．つぎにAという名の最新のマクロをマクロ定義表で引くことになる．今定義したばかりのマクロが見付かり呼出しの結果は文字Bである．これ以前にAという名のマクロがあっても無視されることに注意しよう．

呼出すべきマクロを実引数の中で定義するこの技法を用いて一種の条件文の効果を出すことができる．x, y, v, w を文字列として

IF $x = y$ THEN w ELSE v

という条件文と同等なマクロを書くことができる．GPM ではこれを

#x, #DEF, x, $\langle v\rangle$; #DEF, y, $\langle w\rangle$;;

と書く．この呼出しの評価によってそれぞれ本体 v, w を持った2個のマクロ x と y を生成する（$\langle v\rangle$ と $\langle w\rangle$ の評価とは，両側のかっこを取除いて文字列 v と w をそのまま転写するもの）．つぎに x という名のマクロをマクロ定義表で引く．もし x が y に等しければ，結果となるべき本体は w であり，さもなければ v である．この例では呼出し前には未定義の x というマクロ名を使用していることに注意．さらに x の実引数は決して仮引数によって呼出されることがないことにも注意しよう．実引数の目的は局所的なマクロ x と y を定義するだけである．無用のように見えるがこの使用法は後述しよう．

このほか興味ある特徴はマクロ名がそれ自体，仮引数および（あるいは）マ

クロ呼出しを含むことができる点である．本体を 3.14159 とするマクロ PI を考えよう．するとマクロ呼出し

#DEF,#PI; ＋1,4.14159;

は 3.14159＋1 という名の 4.14159 という本体を持ったマクロを生成する．

Strachey によって与えられた例題の一つにつぎのマクロがある．

#DEF,SUC,⟨#1,2,3,4,5,6,7,8,9,10,#DEF,1,⟨&⟩ &1;;⟩;

これは

#1,2,3,4,5,6,7,8,9,10,#DEF,1,⟨&⟩ &1;;

という本体を持った SUC というマクロを定義する．ここで SUC,2 のようにマクロ呼出しをしてみよう．マクロの本体の走査を始めると直ちに引数 2,3,4,5,6,7,8,9,10 と #DEF,1,⟨&⟩ &1; を持ったマクロ名 1 の呼出しに出会う．まず本体&2 をもった1というマクロの定義が行われる（&の両側の区切り記号が取除かれ&1 が SUC の第1引数2で置換るので&2 が生成される）．つぎに 1 を呼出す．この本体は&2 であるから結果は第2引数 3 である．

一般に #SUC, r; は $r=0,1,\cdots,9$ に対して $r+1$ を生成する．すなわち SUC は与えられた数字のつぎの数字を生成するマクロである．

条件文の技法を用いると 2 桁の数字のつぎの数字を与えるマクロ SUCCESSOR に発展させることができる．例えば，#SUCCESSOR,2,3; の結果は 2,4 であり，#SUCCESSOR,2,9; の結果は 3,0 である．

SUCCESSOR の本体は，

#&2,#DEF,&2,&1 ⟨⟨,⟩ #SUC,⟩ &2 ⟨;⟩;
#DEF,9,⟨#SUC,⟩ &1 ⟨;⟨,⟩0⟩;;

である．i,j を10進数として #SUCCESSOR i,j を呼出してみよう．実引数 i,j を評価しても，マクロ呼出しに出会わない．マクロ本体を走査するとマクロ j（これは第2実引数）の呼出しが生成される．本体のただ一つの実引数は，まず

マクロ名 j　　本体　i ⟨,⟩ #SUC,j;

というマクロを生成し，つぎに

マクロ名9　　本体　#SUC, i; ⟨,⟩ 0;

というマクロを生成する．ここに条件文の技法が使われている．さてここでわれわれが呼出したマクロ j を探そう．もし j が9でなければマクロ j の本体は i ⟨,⟩ #SUC, j; である．これを走査すると結果は

i, k (ただし $k = j+1$)

となる．もし j か9であれば，マクロ9の本体は #SUC i; ⟨,⟩ 0 だから

n, 0 (ただし $n = i+1$)

が生成される．このようにして #SUCCESSOR, 2, 3; は 2, 4 となり #SUCCESSOR, 2, 9; は 3, 0; となり，#SUCCESSOR, 9, 9; は 10, 0 となる．

この例は GPM の機能のほとんど全てを説明している．すなわち，仮引数となるマクロ名，必要な時まで評価を後回しにする文字列の区切り記号の用法，局所的なマクロ定義の用法，呼出しを開始した時点では未定義のマクロ呼出しの使用法，それに条件文の利用などである．

議　　論

嬰記号 (#)，セミコロン，コンマおよび&が意味を持たないプログラム用言語である限り，GPM はその言語の前処理部として使用できる．実際これは Titan 計算機 (Atlas 2) のアセンブリ言語の場合にそうであった．この計算機で GPM は初めて作動したのであった（そこで使用された記号は，今ここで使ったものと同じものではない）．文法も幾分制限されたものであるがこれによって計算機での実現が幾分簡単になることが Strachey に書かれている．

先に挙げた例 SUC や SUCCESSOR の価値に疑問をさしはさむ人もあるいはあるかもしれない．これらの例がこれまでにプログラム中で使われたかどうかは知らない．しかしこれらの例はどんなことを行いうるかを巧みに説明している．GPM 風のシステムの実用性はアセンブリ言語でコンパイラを書くとき実際それを利用してみて説明できる．現実のマクロ処理系はそれ自体かなり遅いものであるが文字中心の凝ったマクロ処理系はどれもそんなものである．つまり1文字ごとに文字の挿入を行うのは時間のかかるものである．

演習問題 19.3

1. GPM を作成するために流れ図を描け.

2. GPM を作成せよ.

19.4 参考文献概説

マクロは最初アセンブリ言語の一部として作成された. それは1950年代の後半にまで逆のぼる. Greenwald (59) には, IBM 709 の SHARE 709 システムにマクロを作成する方法が述べてある (Greenwald と Kane (59) も参照). このシステムは行単位であり, 大抵のアセンブリ言語はそうである. McIlroy (60) は, マクロの使用を広めるのに大変役立った. この論文は, 条件付マクロ, 内成記号, 実引数の部分リストの使用, 入れ子になった定義と呼出しなどを述べている. これには作成法も簡単に述べてある. Kent (69) はマクロの使用法に関する見事な解説書である.

高水準言語の中に埋込まれてその言語を拡張する役をするマクロの使用は McIlroy (60) に議論されている. つまりこの着想は昔からあった. B5500 ALGOL では引数なしのテキストの純粋な置換えを許している. FORTRAN II には引数付きの特別な形式のマクロがある. すなわち

```
FUNCTION F(A, B) = A*B*SIN(A)
```

と宣言しておいて

```
Q:= F(2*G, H)
```

と書くと, 実際には

```
Q:= 2*G*H*SIN(2*G)
```

という文があるのと同じようにコンパイルされる.

19.2節では PL/I である種のマクロ機能をどのように実現するかを示した. コンパイル時の文と PL/I の文は%記号を除けば全く同じである. したがって, マクロ機能と言語とは非常に密接に関連している. Strachey の GPM のように言語とその中で使われているマクロ機能とが全く独立なシステムもある. McIlroy (60) はこの考え方は Perlis によると云っている. このようなマクロ処理系を最初に論文にしたのは Strachey (65) である. それ以来, 言語と独立したいくつかのマクロ処理系が設計された. 例えば TRAC (Mooers と

Deutsch (65)), Waite (67) による LIMP, Brown (67) による ML/1 マクロ処理系などである．ML/1 は本質的にはマクロ呼出しを開始させるキーワードによって動作する．これは使用者に実引数の区切り記号と呼出しの終止記号を定義させるようにしている．Waite のシステムでは，キーワードは必要ない（使うこともできる）．ここではマクロ呼出しを検出するのに一般的なパターン一致機構を用いている．

Halpern (64, 67, 68) は，高水準言語の基礎として巧妙なアセンブリ言語マクロシステムを使うことを提唱している．高水準言語で書かれた原始プログラムは，全て実際にはアセンブリ言語のプログラムとする．すなわち高水準言語の原始プログラムの各文はそのアセンブリ言語で定義されたマクロの呼出しなのである．例えば

```
DO 10 I = 1, N
```

は実引数 10, I, 1, N を持った DO というマクロ呼出しであり“＝”と“,”は実引数の区切り記号である．

この方法はある種の言語を計算機で作成するには良いかもしれないがアセンブリ言語であるために一般に行単位処理である．さらに ALGOL の様なブロックの入れ子や条件文などの処理をやってのけるのは難しい．

マクロ処理系に関するより一層詳しい解説は Brown (69) にある．

第20章

トランスレータ作成システム

20.1 序　　論

トランスレータ作成システム (translator writing system; 略して **TWS**) とはトランスレータ——例えばコンパイラ，インタプリタ，アセンブラなど——の記述が容易なプログラムあるいはプログラム群（どんなものでも）である．この定義によれば，生成言語コンパイラ（第7章参照）や自動構文解析アルゴリズム用の自動作成機も TWS である．TWS の目的は，もちろん，トランスレータの作成実現を容易にすることで，このため TWS には，大部分のトランスレータが実行しなければならない各種の操作に対する基本的な道具が含まれる．

TWS の中で行われる仕事は，特にコンパイラを書くことを指向することが大部分であったので，**コンパイラ-コンパイラ** (compiler-compiler; 略して **CC**) という術語がこの種の TWS に対して定着している (Brooker, Morris および Rohl (62), Brooker ほか (63) 参照)．この術語の由来は，コンパイラを書くために設計された TWS が，他のコンパイラをコンパイルするプログラムであることによる．同様にして，あるコンパイラを記述しているアセンブリ言語のためのアセンブラをコンパイラ-アセンブラと呼んでもよさそうなものであるが，そうはいわない．

本章では，TWS の概要を簡単に述べ，普通に行われている TWS の分類を概観する．つぎに特に二つの CC に注意を向け，これらの働きをもっと詳細に述べる．本章は TWS の概説を行うものではなく，TWS の簡単な紹介である．Feldman と Gries (68) を参照すれば，TWS の調査報告と，多数の文献がのっている．

TWS で書かれたコンパイラは，アセンブリ言語で書かれたそれと同等のコンパイラに比べて，大抵の場合，記憶容量を余計に食い，プログラムをコンパイルするのがずっと遅くなる．例えば，Brooker, Morris および Rohl (67) は，1年のうちに，コンパイラをアセンブリ語に書変えて，容量で 1.6，時間で 1.7 の比率で減らすことができた．それにもかかわらず，長い目で見れば，高水準言語や TWS は，コンパイラを書くための道具としてアセンブリ語に，きっととって代るだろう．それはちょうど，大部分の科学技術計算のプログラムの記述がアセンブリ言語から高水準言語にとって代られたことを見ればよい．コンパイラを高水準言語で書けば，こみいったプログラム上の詳細な点にアセンブラ言語の場合のように気を使う必要がないので，コンパイラが当然なすべき仕事に注意力を集中することがずっと容易になる．例えば，Lowry と Medlock (69) は，IBM FORTRAN H コンパイラを FORTRAN ではなくてアセンブリ言語で書いていたとしたらこのコンパイラのコードの最適化機能の全てを実現できたかどうか疑わしいと述べている．

TWS の背後にあるねらい

大抵の TWS は，少くともつぎの二つの要素から構成されている：

1. 翻訳されることになっているデータ（でき上ったコンパイラの入力となる原始プログラム）の構文を記述するための言語の部分．
2. 意味付けルーチンを記述する言語の部分（これはすでに述べたようなもの）．これは，手続き記述言語，例えば，アセンブリ言語，FORTRAN, ALGOL などである．

トランスレータは，構文の記述と意味付けルーチン群から構成されている．**図20.1** に描かれているように，これら二つの要素は，まず TWS によって機械語あるいは，適当な内部形式に翻訳（コンパイル）される．この過程は**メタコンパイル時** (meta compile time) に行われる．結果として得られたトランスレータがつぎに実行可能となり，この実行が，通常の**コンパイル時** (compile time) である．この実行は，構文の記述と入力される原始プログラムによって制御される．意味付けルーチンは，それぞれのルーチンに関連する構文要素が認識されるにつれて実行される．

TWS のかなりの部分は，基本的なサブルーチン群からなっている．これらのルーチンはどんなトランスレータも自動的に使用するものである（図20.1

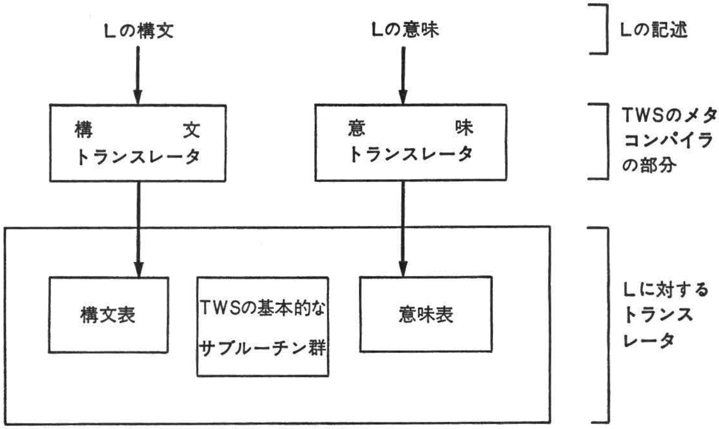


図 20.1　TWS の構成

参照)．すなわち標準的な I/O ライブラリ，算術関数，そのほかの標準ルーチンで算法言語で用いられているものと同等のものである．ところが，ここにも，これらの多くがあって，トランスレータが必要とする機能を遂行している．

TWS において構文の記述をコンパイルする部分は**自動作成機** (constructor) である．この用語は構文の章 (第6章) で定義した．入力データは，普通，BNF 型をしており，自動作成機は構文解析アルゴリズムを，実行または解釈実行できるようなプログラムの形式で生成するか，さもなければ，いくつかの表を作り出す．これらの表は，構文解析を行うためにライブラリ中に用意された基本的なルーチンで用いられる．

図 20.1 で，“原始言語 L の意味” という言葉を使っていることに注意．この言葉は，いくつかの TWS が用いているものであるが，極めて誤解を招き易い．何となれば，これは，構文の記述というのと同じような用法での意味の記述といっているのではないからである．これは，ユーザが，L の意味を理解するために，見ることが可能な記述ではなくて，まさに手続き記述言語で書かれた意味付けルーチン群そのものなのである．これらのルーチン群は，実行される時，原始プログラムの意味の解析を行い，原始プログラムを，別の形式に変換することを目的としている．意味用の言語には，ほかの手続き用言語にも

備っている以下に示すような通常の文法要素があるのが普通である：

1. REAL, INTEGER, BOOLEAN, STRING 型
2. 単純変数，配列，表，（多くの場合）構造体（あるいはレコード）
3. 代入文，条件文，繰返し文
4. 入出力文

さらに，コンパイラを記述するのに役立つ道具立てがこのほかにもある：ポインタ型変数，スタック，表にエントリを作り索引するための基本要素，コード生成用の基本要素など．

大抵の TWS は，1パスコンパイラしか生成しない．これは TWS の本来の機能に由来する制限ではなく，大部分の TWS がわずか半年の講義の中でなんらかの着想とか概念を試すために設けられた演習問題として作成される場合が多かったからである．パスが複数のものもいくつかあり例えば，Cheatham (65) や AED システム (Ross (67a) と Ross (67b)) などを参照されたい．

TWS の分類

CC（コンパイラ-コンパイラ）は TWS の一つの類を形成している．CC はコンパイラを作成することを指向しており，コードの最適化や，記憶割付などに役立つ基本的な機能を持っている．

TWS の第2類は，構文向きの記号処理を行うものである．この類に属するものはより一般的で，文字式の簡単化，文字式の微分，ある高水準言語を他の高水準言語へ翻訳，ある書式のデータを別の書式に変換などの仕事に用いられてきた．これらもコンパイラの作成に使用されてはきたが，その出力は普通の場合アセンブリ言語のプログラムでこれは，さらにアセンブラを通してやらなければならない．一般的にいって，これらのシステムが有用なのは，入力の構造 (structure) が，その内容の大部分を含んでおり，しかもこの構造が BNF 型の語句で記述されている場合である．

第3類は，拡張可能言語の集合である．これらの言語では，プログラマは既存の要素を用いて，新しいデータ型や新しい実行文を定義することができる．これによって，プログラマはその言語を自分の必要に合致したものに拡張したり変更したりすることができる．しばらくは，制限付の拡張を認めた言語がいくつか存在していた．例えば，MAD では，新しく挿入(インフイクス)演算子が定義でき

る．真の意味での拡張可能言語ならば，ほとんどどんな構文を持った実行文や演算でも新しく追加することができる．このような拡張は，高水準言語の中にあるマクロの考えと結びつく．これは19章で概説した．Brooker と Morris のコンパイラは次節で簡単に紹介するが，これは，CC と同様に，拡張可能なコンパイラとして使用することができる．

20.2 コンパイラ－コンパイラの紹介

FSL 型言語

Feldman (66) の FSL (Formal Semantic Language) はコンパイラ記述システムであり，Carnegie-Mellon 大学の G-20 と Lincoln Laboratory の TX-2 の二つの計算機の上で作成された．FSL は走査部を定義する機能（3.4 節で述べたものと非常によく似ている）とともに，生成言語（PL，第 7 章参照）を使用する．これは原始言語の構文を定義するためのものである．

コンパイル時には，PL プログラムの形式の構文解析部が記号を保持するためにスタック S を用いる．これと並列に S 中にある構文記号の意味を保持するためのもう一つの意味用スタックがプログラマによって使用できる．生成規則によって呼出された意味付けルーチンは，つぎのようにして意味用スタックを参照する：L1, L2, …, L5 は，生成規則の一致がとれた後，スタックの内容の変換が行われる直前のスタック内の配置を考えたとき，それぞれ意味用スタックの上から1番目，2番目，…，5番目の要素を引用しており，R1, R2, …, R5 はスタックの内容の変換が行われた**後の**スタック内の配置を考えたときそれぞれ意味用スタックの要素を引用している．

例を示せば直ちに明白になろう．もし生成規則

```
E1  +  T ANY → E2 ANY EXEC 5
```

と一致がとれると，意味付けルーチン5の中では，L2 と L4 がそれぞれ T と E1 の意味を参照するが，一方 R2 は E2 の意味を参照する．

FSL は主として意味用言語で用いられるコード生成機構で役立っている．それをここで論じよう．FSL を精しく議論するのではないことを承知しておいてほしい．記号法といくつかの重要でない論点を変更してわれわれの擬 ALGOL と一致する意味用言語に持ってゆこう．

この意味用言語における新しいデータ型は DESCRIPTOR（意味語）で，これはコード生成しつつある**機械のことば**で，実行時の変数を記述するために用

いられる．FSL で書かれたコンパイラはいずれも，一般的なコード生成ルーチン群を用い，これらは実行時変数を記述するのに意味語を用いている．これらのコード生成ルーチン群は，普通のコンパイラに備わっているそれらに対応するルーチン群とそんなにかけはなれたものではない．ただ幾分一般性を持たねばならないだけである．

意味語型のデータは，いくつかの構成要素から成っている．最初の二つは後で名前を付けるが，普通はコンパイラ作成者によって変数を記述するために初期値が埋められる．残りは，ここでは参照することがないので名前は付けないが，コード生成システム自体が値を書きかえたり参照したりするもので，コンパイラ作成者がするのではない．意味語の構成要素にはつぎのようなものが含まれている：

1. ADDRESS：変数の実行時の番地を与える整数．
2. TYPE：コードを生成中の**機械のことば**で変数の型を与える整数（例，1 = FIXED = 固定小数点の語，2 = FLOAT = 浮動小数点の語，3 = BITS = 1語長のビットか符号なし整数）．
3. 間接番地を示す欄．
4. 変数が添字付きか否かを示す欄，もし添字付きなら，添字の値を示す欄が加わる（後述）．
5. 実行時変数にマイナスをつけたものか否かを示す欄（これは，局所的に良いコードを生成するために役立つ）．
6. 記述されたデータが現在，累算器あるいはレジスタ上にあるか否かを示す欄．もしあればそのレジスタを示す欄が加わる．等々．

最初の2欄を除いてあとは全てコードを生成中の実行時の機械の“状態”を記述するのに有用なものである．これらの精しい説明はコードを生成している機械に依存し，特定のコード生成システムにも依存している．

コードを生成するのに意味語がどのように使用されるかを示すために，ブロック構造を持たない適当な言語での宣言，名前，乗算の処理に対する意味付けルーチンを説明しよう．ここで関心のある規則は，

```
(20.2.1)    dec    ::= REAL idlist
            idlist ::= I | idlist , I
            F      ::= I
            T      ::= T * F
```

であり，これに対応する PL における生成規則は

```
(20.2.2)  LDEC : REAL       →              EXEC 1 SCAN GO LID
          LID  : idlist, I  → idlist       EXEC 2 SCAN GO LID
                 I          → idlist       EXEC 2 SCAN GO LID
                                  ...
          LF   : I ANY      → F ANY EXEC 3 GO LT
                                  ...
          LT   : T * F ANY  → T ANY EXEC 4 GO LT1
```

となるだろう．意味付けルーチン 1 は生成規則 LDEC で呼出される．このルーチンの唯一の機能は，それにつづく名前の並び (idlist) の中にある名前の型を，大域的な変数 T に設定することである．同様なルーチンは，INTEGER, BOOLEAN, などに対しても存在する．

1: T:=FLOAT;	FLOAT は浮動小数点数であることを示す値を持った変数である．

意味付けルーチン 2 は宣言された名前がスタックの最上段にあるときに呼出される．したがって，このルーチンはこの名前を L1 で参照する．ここで記号表 SYMB がこのコンパイラで使用されると仮定する．ルーチン 2 は，この名前を探索してそれがまだ表中にないことを確かめる（簡単のため，名前は使用される前に宣言されてなければならないことにする)．つぎにこの名前を，その属性とともに SYMB にのせなければならない．

SYMB のエントリの各欄の一つに意味語 DESCR があり，これは，変数の実行時の特性を記述するものである．DESCR の TYPE 欄も，この変数の原始プログラムでの型を与えるために用いる：FIXED は INTEGER を意味し，FLOAT は REAL を意味する．場合によっては，原始プログラムの型を記述するためには記号表に余分な欄を設ける必要があろう．例えば，整数値と実数値がともに浮動小数点数で実現されていても，コンパイラは依然として名前や式のそれぞれの型を知って適宜に型変換の演算を遂行したり生成したりしなければならないのである．

ルーチン 2 は，意味語を大域的変数 T の値に応じて，FLOAT 型か FIXED 型で組立てる．STORLOC は，実行時変数に番地を割付けるのに使用されるコンパイラでの変数である(初期値は 0)．意味語を作り上げると，システ

ムは参照してない欄には自動的に適当な初期値を埋込み，未定義でなくする．

2: LOOKUP(L1, P);	名前を記号表で引いて．
IF P ≠ 0 THEN ERROR(1);	もしあれば二重定義で誤り，なければそ
INSERT(L1, P);	れを表にのせる．P. STRUC=1は，そ
P. STRUC:=1;	れが単純変数を意味する．
D:=DESCRIPTOR(ADDRESS=	実行時変数のDESCRIPTOR(意味語)
STORLOC,	を作成し
TYPE=T);	
P. DESCR:=D;	これを記号表に．
STORLOC:=STORLOC+1;	実行時の番地カウンタを1あげる．

ルーチン3は，名前 I が因子に還元されるときに呼出される．このルーチンは，記号表を引いて，F の意味として意味用スタックの上に関連する意味語をのせる．P. STRUC = 1 の検査は，これが単純変数であることを確かめるためのものであることに注意．配列名，手続き名，などは P. STRUC に別の値が入っている．

3: LOOKUP(L2, P);	名前を記号表で引く．
IF P = 0 THEN ERROR(2);	表中に見つからなければ誤り．
IF P. STRUC ≠ 1 THEN ERROR(3);	単純変数でなければならない．
R2:=P. DESCR;	意味語をスタック上にのせる．

大部分のコンパイラでは，乗算のためのコードを生成するために，MULT (D1, D2, D3) のようなコード生成手続きを呼出す．ここで D1 と D2 は乗算の実行時の被演算子であり，D3 は結果の記述をしまうために用いられる．FSL では，このようなコード生成用の文は，

```
D3 := CODE(D1,D2);
```

のようになっている．意味付けルーチン4は，規則 T ::= T*F に対するコードを生成するものである．L2 と L4 がそれぞれ F と T に対する意味用スタックの要素を参照するものであり，これらの被演算子の各意味語である．コード生成手続きから得られる意味語は R2 にしまわれる．これは，左辺の非終端記号 F に対応する意味用スタックの上にある．

4: R2:=CODE(L4 * L2);	

FSL では，ほとんどどんな実行文や式の系列でもコード用かっこ "CODE (" と ")" の内側に現れる．このようなコード用かっこに囲まれた系列がコンパイル時に実行されると，その系列に対するコードが生成される．コード用かっこ内の被演算子が，意味語か定数でなければならないのが主たる制約である．例えば，A, B, C と I が実行時変数で，コンパイル時の意味語 DA, DB, DC と DI でそれぞれ記述されていると仮定しよう．

(20.2.3)　　CODE (DI := DA + DB * DC)

を実行すると，実行時に代入文 I := A + B * C が実行されるようなコードの生成を引起す．

コード用かっこが式を囲んでいる場合，この CODE 文を実行した結果は，コードが生成されるばかりでなく，結果の意味語も生成される．この意味語は（意味付けルーチン4でしたように）適当な場所にしまっておいて後で用いるのが普通である．別の例として

DI := CODE(DA + DB * DC) ;

を実行すると A + B * C を評価するコードが生成され，（実行時の）結果が累算器かレジスタの上に残される．結果を記述しているコンパイル時の意味語は変数 DI にしまわれている．

FSL のコード生成で用いられる表記法は，普通のコード生成の表記法よりもずっと見事で，読み易い．コード用かっこに括られていることを除けば，普通の実行文と変らない単一の実行文の形をしており，コード生成用ルーチン群を呼出す一連の命令が並んでいるのではないからである．

FSL は実行時の添字計算に対してもすっきりした表記法を用意している．生成規則

I (E) → F　EXEC 5

に対する意味付けルーチン5を記述したいものとしよう．ただし I は配列名で（その意味語は L4 にある）E は式である（その意味語は L2 にある）．I の意味語の ADDRESS 欄（L4.ADDRESS）は実際には配列のドープ・ベクトルの番地で，実行時ドープ・ベクトルの第1語には CONSPART の値が

入っている (8.4 節参照). つぎには, 実行時変数の番地が CONSPART ＋ (E の現在の値) であることを明記する意味語を生成したい. これをつぎの2段階で行う.

D1 := ＊L4;	L4 は意味語. この代入文は, 間接番地を指定しているところを除いて L4 の成分と同じ成分を持った意味語を生成しそれを D1 にしまうことを示す. かくてその実行時番地は CONSPART の値である.
R1 := D1(L2);	記述された変数の番地が (意味語 D1 中に記述された実行時番地) ＋ (意味語 L2 で記述される実行時の値) であることを除けば, D1 の成分と同様の成分を持った意味語を生成しそれを R1 にしまうことを示す.

R1 にある意味語は添字付変数を記述している. 配列要素の番地を正しく計算するコードを生成するのは, コード生成システムの仕事である. 番地計算の過程は, 通常の番地計算に非常によく似ていて, 違いは通常のコンパイル時変数の代りに意味語が入ってくるだけである. 上で二つの代入文で書いたものを実際には単一の代入文 R1 := ＊L4(L2) とすることもできる.

FSL コンパイラにあるコード生成用の実行文はメタコンパイラによって翻訳されて通常のコード生成ルーチンを呼出す一連の機械語命令に変換される. 例えば, 実行文 (20.2.3) は

```
MULT(DB, DC, DT1)
ADD(DA, DT1, DT2)
STORE(DT2, DI)
```

と等価な機械語命令に翻訳されることになろう. ただし MULT, ADD, STORE は, それぞれ, 乗算, 加算, 代入に対するコード生成ルーチンである.

読者は, 構文解析アルゴリズムが呼出す, もともとの意味付けルーチンの中でコードを生成しなければならないと想定する必要はない. 意味付けルーチンは, 普通どおり内部形式を生成し, コードはそのあとでコード用かっこを用いて, 内部形式から生成するというのが, 融通性のあるシステムである.

BMCC

BMCC とは Brooker-Morris のコンパイラ－コンパイラの略称で，最も古い TWS の一つである (Brooker と Morris (62), Brooker ほか (63), Rosen (64) 参照). さらに，これは実際に生産的に用いられている数少い TWS の一つでもある.

構文解析のために，BMCC はシステムに標準的な走査部を備え（これはコンパイラ作成者が自分で作ったものと入換えることが可能である）下向き解析法を使用している. 左再帰性は，下向き法のもとでは不都合なので，これを置換えるために，構文の記述には，特別な記号と繰返しを用いるようにさせている. 文法の各規則に１対１の意味付けルーチンを関連させる必要はなく，またそうしないのが普通である. 例えば，代入文に対するつぎのような構文の記述(不完全である) を考えてみよう. ただし右辺の式は一連の加減算のみである：

1. FORMAT [SS] ::= 〈変数〉 : = 〈和〉
2. PHRASE 〈和〉 ::= [〈符号〉] 〈項〉 〈項々〉
3. PHRASE 〈項〉 ::= 〈変数〉 | (〈和〉)
4. PHRASE 〈項々〉 ::= 〈符号〉 〈項〉 〈項々〉 | 〈空〉
5. PHRASE 〈符号〉 ::= ＋ | －

構文規則の直前にある PHRASE という語は，この規則に関連する意味付けルーチンが存在していないことを示すものである. 最初の規則の直前にある FORMAT [SS] は，原始プログラム実行文 (Source program Statement) が〈変数〉: = 〈和〉という形式で，この形式の実行文が構文解析されるときには，それに連らなって実行されるべき意味付けルーチンが存在することを示している.

コンパイル時に，原始プログラムは構文解析されつつ，構文の木が形成されてゆく（したがって，下向き解析プログラムは原始プログラムの内部形式を生成する). FORMAT [SS] に続く構文規則（上例の〈変数〉: = 〈和〉など）に対応する部分木が形成されるとそれに対応する意味付けルーチンがつねに呼出される. 構文解析された部分木は，この意味付けルーチンの引数として渡される. このルーチンの目的は，形成された部分木をいつものように意味的に処理するものである. この場合，このルーチンは，部分木の上を“歩き回り”これに対するコードを生成すると仮定されている. 上にあげた規則を用いて形成された部分木を処理できるルーチンを以下に示し，これをしばらく議論しよう.

```
1.  ROUTINE [SS] ::= 〈変数〉 : = 〈和〉
2.        LET 〈和〉 =[〈符号〉] 〈項〉 〈項々〉
3.        AC : = [〈符号〉] 〈項〉
4.  L1 : GOTO L2 UNLESS 〈項々〉 = 〈符号〉 〈項〉 〈項々〉
5.        AC : = AC 〈符号〉 〈項〉
6.        GOTO L1
7.  L2 : STORE AC IN 〈変数〉
8.  END
```

意味用言語には，パターンの一致（木の一致）を調べるための実行文がいくつかある．構文の記述の中にある非終端記号はいずれも，変数として使用でき，その変数は，根にその記号の名が付けてある部分木を，値として持つものである．例えば，上にあげたルーチンの第1行を考えてみよう．このルーチンに引数として渡された構文の木の根は

FORMAT [SS] ::= 〈変数〉 : = 〈和〉

に対応するものであることを思い出そう．意味付けルーチンへの名札づけを除けば，第1行は，変数〈変数〉に節点〈変数〉を根とする部分木を代入し，変数〈和〉に節点〈和〉を根とする部分木を代入することを述べている．したがって，もし構文解析された代入文が I : = A + B であれば，〈和〉には図20.2 の最も左側の木 (a) が代入されている．

さて，部分木〈和〉の根からでる枝は，図20.2 に描くように

〈和〉 ::= [〈符号〉] 〈項〉 〈項々〉

という規則を適用することに対応している．ルーチンの第2行目を実行すると，木 (b) を〈項〉に代入し，木 (c) を〈項々〉に代入する．〈符号〉は現れていないし，なくてもよいので，これには〈空〉が設定される．3行目の実行文は，最初の〈項〉を累算器（AC）にのせる目的コードを生成する．生成されたコードは，今〈項〉に代入された部分木に依って異なる．この実行文については，さらに後程説明を加えよう．

4行目の実行文は，変数〈項々〉にある部分木が〈符号〉〈項〉〈項々〉という形式になっているかどうかを調べるものである（これ以外の形式としては〈空〉がある）；もしこの形式であれば，3個の変数，〈符号〉，〈項〉，〈項々〉に，新しい値を与えて，つぎの実行文を実行する．この形式でなければ，制御

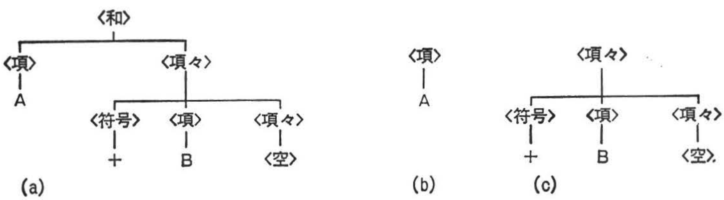


図 20.2 LET 文の説明

は7行目の実行文に移る．5行目と7行目の実行文は現在〈符号〉,〈項〉,〈項々〉に代入されている値としての部分木に基いてコードを生成するものである．

一般に，構文の記述中に現れる構文要素としての非終端記号は，変数として使用できて根にその記号の名前を持った部分木だけがその値となりうる．木の上を歩き回ることは極めて複雑で，同じ名前の根を持ったいくつかの部分木を同時に処理することが必要となるので，このシステムには，各非終端記号に対して変数を要素とする配列を一つづつ持たせている．構文要素〈項〉に対しては，〈項/1〉,〈項/2〉,〈項/3〉, ･･･,〈項/A〉のようにして参照する（ただし A は整数型変数）．この機能は上の例では必要なかった．

BMCC には言語の拡張に都合のよい機能がいくつかある．BMCC で書かれたコンパイラは，ある程度までは実際 BMCC 自身を拡張したものである．この拡張性のゆえに，実行は極めて遅くなる．このシステムは絶えず構文の木を組立ててその上を歩き回っている．これは原始プログラムの文に対してばかりでなく，他のものに対しても同様である．

コンパイラを書く第1段階は，そのコンパイラを実現するために使用する実行文をいくつか新しく設計することといってもよいかもしれない．例えば，先に述べた意味付けルーチンの3行目，5行目，7行目のようないくつかのコード生成用実行文を加えることができよう．これらの三つの文は実は，はじめから BMCC の一部であったのではく，このコンパイラ作成を容易にするために加えられたものである．

BMCC に新しく基本的な実行文の集合を加えて BMCC の拡張版 BMCC をつぎのように形成する．新しい実行文のそれぞれについて，構文の記述を書

き FORMAT [SS] の代りに

FORMAT [BS]

を用いて規則を記述する．この規則にはそれに関連する意味付けルーチンがある．BS は基本文 (basic statement) の略称で，SS は原始プログラム文 (source program statement) の略称である．同様にして，意味付けルーチンを書き，これは，上の形式を持った基本文が実行されるべきときにはつねに実行される．意味付けルーチンの第1行目には ROUTINE [SS] の代りに

ROUTINE [BS]

を用いる．例えば，AC := AC〈符号〉〈項〉に対するルーチンは累算器に〈項〉を加えるか減ずるかいずれかのコードを出力するだろう．もし，この〈項〉が単一の変数であれば，これは，単一の命令になる．もしこれが (〈和〉) の形をしていれば，部分木〈和〉に応じて，一連の命令となる．意味付けルーチンは，全て BMCC のの実行文と，さらに，このような拡張によって追加されたものは何でも使用することができる．

ある意味で，コンパイル時に，AC := AC〈符号〉〈項〉を実行することは，まさに，ある手続き――意味付けルーチン――を呼出すことで，ただ呼出しに都合のよい形式をしているだけである．

構文の記述と意味付けルーチンが BMCC に投入され内部形式に翻訳される．この内部形式は，BMCC の写しと一しょになって新しいメタコンパイラ BMCC 1 が生成される．BMCC 1 は，普通の BMCC の文と今追加された新しい基本文を受理する．

つぎの段階は，原始言語の各文に対する構文の記述と意味付けルーチンを書くことである．これらは，FORMAT 文と ROUTINE 文で BS の代りに SS を用いて記述される．記述したものを BMCC 1 に投入すると，3番目のコンパイラ BMCC 2 が生成される．このコンパイラは，普通の BMCC の文，BMCC 1 に拡張の段階で追加された基本文とそれに新しく定義した原始言語の文を受理する．この点でなお BMCC 2 は拡張可能なコンパイラで，あと何回でも拡張することができる．これを，原始言語の文だけに対するコンパイラに変換するためには，SS として宣言された文だけしか，正常な入力として認めないことを示すスイッチを設定する．これによって，原始言語の文だけに対するコンパイラが得られる．

第21章

コンパイラ作成者への指針

本章には，種々な話題に関する注意が含まれている．これらは，独立した節や章を設けて論ずるにはあまりに小さすぎたり，非常に重要なのに大きな章の中では目立たずにやりすごしてしまったり，あるいは，どの章にも属さないようなものである．注釈は簡単にして，読者が自分で文献に当ってもっと詳しく調べるように指示している．時々この章を通読しては，現に勉強している主題に触れる点があれば，そこを熟読することを奨めたい．

本章で述べる話題をその順に列挙する：

一般的な考え方
コンパイラはどんな言語で書いたらよいか
構文解析を助ける走査部
構文解析アルゴリズムに関する議論
意味付けルーチンに関する指針
コンパイラの構造
コンパイラで用いる表の構成
再入可能コード
コンパイラのデバッグ
コンパイル時のエラーメッセージ
ブートストラップ

一般的な考え方

"コンパイル時にできることを実行時まで延ばすな"

コンパイル時にする操作は，一度だけ行えばよい．しかし実行時まで持越すと

何百回も実行しなければならないかもしれない．このようなものに主として定数の変換，型の検査，定数や実行時に値が既知となる変数を含む演算などがある．どんなコンパイラでも相応の最適化は行うべきである．その程度は，もちろん，コンパイラ作成計画の目的に合せるが，このことを少しでも考えておけば，考えない場合より優れたコンパイラができ上るだろう．

コンパイラはどんな言語で書いたらよいか

当然のことながら，アセンブリ言語から次第に高水準言語が選ばれる傾向にある．理由は明らかで，プログラミングが容易で，デバッグが少なくて済み，費用が安くなり，よいコンパイラを作るためにそれだけ余計に時間を費やすことができ，レジスタの選択やビットの割付に思いまどうことが少なくなるからである．

どの高水準言語を使ったらよいかということは，どの言語が使用可能なのか，コンパイラ作成者が使い慣れた言語が何かに依ることが非常に多い．IBM の FORTRAN H コンパイラは FORTRAN で書かれ，Burroughs 社の拡張 ALGOL コンパイラは拡張 ALGOL で書かれた．“システムプログラム用”言語がもっと発展することを私は信じている．このような言語は高水準であるが，特定の機械向きになっている．これによって，コンパイラ作成者が自分のコンパイラで生成したコードに，より目を利かし，より一層能率のよいプログラムを作ることができるようになる．他方，PL/I のような汎用言語を用いる方法もある．これによって，コンパイラの移し換えは容易になるが，能率は悪くなる．

構文解析部を助ける走査部

走査部で少し余計に仕事をすれば構文解析部の仕事をずっと単純で容易にすることができる．例えば，定数は走査部の中で変換を終えておけば，定数の構文は構文の記述の中には含めないですむ．また走査部ではなんらかの誤りの検査や回復（あるいは訂正）を行うことができる．例えば定数の記述の誤りとか ** のような2重文字記号中の空白とか，文字列定数における区切りを示す閉じ記号の脱落などである．しかしながらこれらの余分な仕事にあまりのめり込むべきではない．走査部はやはり単純で能率がよいことが大切である．

構文の記述には，いくつかの記号に対して前後の文脈が同じものがしばしばある．例えば，＋と－は同じように用いられるし，ALGOL では，REAL

と INTEGER の記号は同じ場所に現れてよい．このようにいわば同類の終端記号全てに対して単一の構文記号を返すように走査部を仕立て，意味ルーチンが実際にどの記号だったかの面倒を見るようにするのが常識的な工夫である．(これは7章の PL でも CLASS 宣言によって行っていた.) 例えば，ユーザにとっては文法には

〈 型 〉:: = REAL|INTEGER|BOOLEAN
〈宣言〉:: = 〈型〉〈名前の並び〉
〈 E 〉:: = 〈T〉|〈E〉+〈T〉|〈E〉−〈T〉

のような規則が含まれているとしても，構文解析部の内部では，これらは

〈宣言〉:: = TYPE〈名前の並び〉
〈 E 〉:: = 〈T〉|〈E〉PM〈T〉

で置換わっている．走査部がすでに名前に対して上のサービスを終えていることに注意しよう．構文記号Iで全ての名前の類を代表し，意味付けルーチンが，個々の名前を弁別する．

ある文法がある特定の構文解析アルゴリズムに適合しないときは，文法を思い切って変更しようとするよりはむしろ，走査部にこの問題の処置をさせる方が容易なことが多い．例えば，ALGOL で単純順位法を用いるときに生じる問題がコンマに関する規則はずれで，コンマの使用法が実引数の区切りであったり，配列宣言の下限上限の区切りであったりすることである．ところが走査部にとって今配列宣言を走査しているかどうかはわかっており，下限上限を分かつコンマを全く異なった記号に変えてしまうのは極めて簡単なことである．

また FORTRAN では，実行文の最初で DO10I を走査したときが例となる．これは，代入文か DO 文のいずれかの可能性がある．走査部が先を読んでいずれかを決定するのは，形式的な構文解析アルゴリズムを用いるよりも容易にできる．

構文解析アルゴリズムに関する議論

Griffiths と Petrick (65) によって各種の下向き解析法や上向き解析法の比較が行われているが，これらの比較の結果は，どの方法がコンパイラの中でより速く働くかを知るという観点からみると実際には役に立たない．アルゴリズムがどのように実現され，意味がどのように適合しているかにあまりに多く依

存している．構文解析法を除いて二つのコンパイラをすっかり同じように作ってコンパイル時に有意な差を測ろうとすることには疑問がある．もちろん，一方のコンパイラの作成が不合理な方法で行われていないとしてであるが．

もっとずっと重要なことは構文解析アルゴリズムの使用に融通性があること，変更を行い，意味付けルーチンを追加することが容易であること，それにそのコンパイラをプログラムし検査するのにかかる時間の3点である．この観点から，5.5節で説明したように，単純順位法が最悪の方法である．第5章全体をこの方法で貫いた唯一の理由は，上向き解析法の概念になじませるのにはこの方法が最も単純で，したがって最も容易であるからである．

もしコンパイラが，再帰的手続きを許した言語で書かれていると，4.3節に述べた再帰的下向き法を必ず考えるだろう．形式的なところも自動的なところもないけれども，手続きを文法に従って書下してゆけばよく，この方法は単純，明解で，デバッグもかなり容易である．最も広く用いられている上向き解析法は，順位関数を用いた演算子順位法である．この方法を用いるコンパイラは，文法から関数を生成するのを機械的にやらずに直観的に行っているのが大部分である．これらのコンパイラでは，6.1節で述べたように2個のスタックを用いるのが普通である．一つは，順位関係にある演算子用で，もう一つは，名前，定数などの被演算子用である．演算子順位法は，式を構文解析するときだけに用いることが多く，事実，この方法は最初この目的のために設計されたものである．これは能率がよく融通性に富んでいる．

形式的な構文解析アルゴリズムは必ずしも使用すべきではない．今強調した2通りの方法は普通は，直観的な方法でプログラムされていることに注目しよう．本書の約三分の一を構文の理論に費やした理由は，一たんこれらの形式的な方法を理解すれば，コンパイラをプログラムし，コンパイラを論じることがより容易になり，より系統的になることにある．特別な技法を用いて容易に構文解析できる言語としてFORTRANは好例である．どんな自動構文解析法にも合った構文の記述を与えることは困難で，さらにこの言語の構造はかなり単純である．ブロック構造もなく，条件文が入れ子になることもなく，CASE文もないなど．FORTRANは行が基本で，行の上の文の種類は，最初の1～2記号で決定できるのがほとんどである．

原始言語の部分部分に応じて別々の構文解析アルゴリズムを用いるのがしばしば有利である．まだ実現していないものの，文法を分析し，これをいくつかの部分に機械的に分割し，各々の部分に対して最も能率のよい構文解析アルゴ

リズムを作り出すような自動作成機を設計する議論がこれまでに行われてきた．これらのアルゴリズムは必要に応じて互に呼出し合うこともする．

Conway (63) は COBOL に対する“分離推移図”コンパイラに言及している．ここでは，言語は，いくつかの SECTION に分けられており，各々は有限状態オートマトンによって構文解析される．これらのオートマトンは，互に再帰的に呼合う．Tixier (67) は独立にこの方法を形式化したが，彼の形式化は，現実のコンパイラには用いられていない．

いくつかのコンパイラは解析を二つの部分に分けている．例えば，FORTRAN H (IBM(a)参照) は，全ての宣言文と実行文を構文解析するのに特別な方法を用いるが，全ての式は，(上で述べたように) 演算子順位法を用いた単一のサブルーチンによって解析される．

意味付けルーチンに関する指針

サブルーチンへの入力となるようなものは全て，ルーチンの仮引数にするというのがプログラミングの要諦である．これによって，一般的で，独立なサブルーチンができ上るが，コンパイラ作成にあたって，大域的な変数 (FORTRAN では COMMON) を多用することがしばしば好都合である．これを，コンパイラ作成の好例の一つで説明しよう．

名前や式を処理する意味付けルーチンは全て，型の一致を調べたり，エラーメッセージを印刷したり，最適化の可能性を検査したりなどのルーチンを少くとも一つは呼出すだろう．これらのサブルーチンはそれぞれ引数として，この名前または式を記憶している場所へのポインタを持っていよう．しかしながら，コンパイルの任意の時点で，そのような被演算子が処理されているのは高々 1～2 個であることに注意しよう．例えば，2 項演算を処理する意味付けルーチンは，処理すべき被演算子は 2 個である．実引数を処理する意味付けルーチンも 2 個で，それは実引数自体とそれと比べなければならない仮引数とである．

したがって，1 (あるいは 2) 個の被演算子を処理する意味付けルーチンはそれぞれ，その被演算子 (あるいは，それを記述している場所へのポインタ) を大域的な場所 P (2 個あればさらに P1) に移すという便法を採用しよう．すると，被演算子を処理するルーチンはいずれも大域的な場所 P と P1 を通じて，被演算子を参照できる．意味付けルーチンが，構文解析アルゴリズムに戻る直前にポインタ P (場合によっては P1 も) を，スタックの適当な場所に戻しておく．

これによって，サブルーチン呼出しが簡単になり，コンパイル時が，時間と空間の両方の観点からより能率的になり，一般に，よりすっきりしたコンパイラになる．

コンパイラの構造

われわれは，コンパイルの手続きを4段の処理過程として述べてきた．それらは，走査，解析，コード生成の準備，それにコード生成である．Cheatham (65) はこれを6ステップの処理過程と考えている．すなわち，構文と意味の解析を2段に分割し，コード生成を二つの部分――第一はコード生成，第二はそのコードを調整して出力する――に分ける．

このようなモデル化は，コンパイラ作成を教える目的に好都合であるが，必ずしも現実のコンパイラでそのまま用いることはできない．使用する計算機や，原始言語や，作成の目標などに依るところがあまりにも多い．コンパイラ作成者は，自由に思うとおりに自分のコンパイラの構造を決定してよいが，同時に，上に提示したモデルから思い切って構造を変えるには確かな理由を示すべきである．上にあげたモデルは，さまざまな検討を経て設計されたものなのであるから．

複数回パスを行うコンパイラはまず第1回は原始プログラムから内部形式 (4-組，3-組など) を作りだすのが普通である．つぎに，この内部形式の各演算子に対して，機械語命令を生成する．もし言語にいくつかの制限を設ければ，初めの意味付けルーチンが，同時に直接機械語を生成することもできることは明らかであろう．この制限の主なものは，全ての変数が，その使用に先立って，宣言部または規制部に現れていることである．もちろん，意味付けルーチンは，多少複雑にならざるを得ないし，生成されたコードは，良いものではないだろうが，このような欠点も，コンパイル時間の高速化を考えれば，十分補われることが多い．

コンパイラにおけるパスの回数は，計算機の大きさやコンパイラ作成計画の目標を考慮の上，最小に止めるべきである．パスが増えればそれを制御するための余分な仕事，各パスでの準備・後始末などが増え，おそらくコンパイラ作成にそれだけ余計な時間を費やすことになろう．

コンパイラ内での表の構成

表やスタックの種類はできるだけ少なくし，その大きさは，原始プログラム

の大きさに依存するようにする．これによって，あふれの生ずる可能性のある表の数を減らして，コンパイルできる原始プログラムを大きくすることができる．このためには，（上向き解析法を用いるコンパイラの）コンパイル時スタックと目的コード表をできるかぎり利用する．例えば，〈条件節〉に対応する意味スタックの要素の中に，条件文に対する前向きの条件付分岐先の番地をしまっていたし（13.2節），前向きの分岐先は内部プログラムを通じて結合された（13.3節）．同様な考え方は，繰返し文，CASE 文などをコンパイルするのに使うことができる．

コンパイル時に表に新しく要素を追加するときは，**いつも**あふれの検査を行う．コンパイラに“虫がいなくなった”何ケ月も，何年もあとになって表のあふれを検査しないでおいたために生じた誤りを追跡するためにどんなに多くの時間が浪費されてきたことか．

いくつかのシステムでは，プログラマが，自分の仕事で使用する記憶容量を予め指定することが可能である．このようなシステムでは，コンパイルが始まる時点で，コンパイラが使用できる最大の記憶容量を要求して，その一部をコンパイルを始めた原始プログラムの大きさに依存した大きさの表にあてるようにする．このような表には，記号表，スタック，原始プログラムの内部形式用の表などがある．これらによって，小さいプログラムは小さい領域でコンパイルすることができ，大きなプログラムだけが不利になる．

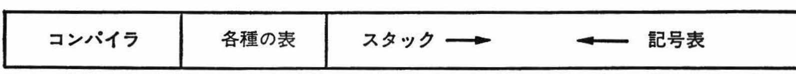
もし，大きさに制限のないスタックや表がたくさん実際に必要ならば，このコンパイラは，これが，組織的に行えるような言語で書かれてなければならない．すなわち，コンパイル時に，このシステムは当然リスト構造を持った記憶に対して要素を表に追加したり，表から削除したりする基本機能を含めた，一般的な記憶割付機構を持っていなければならない．このようなコンパイラは，ブックキーピングが増えるために，どうしても遅くならざるを得ない．

表を記憶するために2次記憶を使用することは最小限にとどめるべきである．小さなプログラムに対してでも表を2次記憶に書出すような費用のかかることをしてはならない．表の書出しはあふれが生じたときだけにとどめるべきである．エントリがランダムに参照されるような表（例えば記号表）は書出すべきではない．2次記憶に書く場合には原始プログラムの内部形式用の表が，最適な候補である．何となればこの表は，逐次的に作成され，参照されるものであり，入出力動作とコンパイルの仕事をある程度まで並列に行わせることが可能だからである．Naur (65) は，非常に小さな計算機の上で走る精巧なコ

ンパイラを書いている.

どうしても2次記憶を使わねばならない場合は，入出力装置とCPUが並列に動作するようにコンパイラを設計すべきである．例えば，プログラムの内部形式をディスク上に作らなければならないのならば，バッファ用記憶場所を設けて最初のバッファに出力情報を満たす．ついで，このバッファの内容をディスクに書出している間に，コンパイラは処理をつづけてつぎのバッファにプログラムの続きの部分の内部形式を生成することができる.

コンパイラの解析部で取扱うデータ構造の主なものは記号表と意味スタックの二つである．アセンブリ言語で書かれたコンパイラでは，記号表は，この仕事に与えられた主記憶の大部分の一方の端から取り，スタックはもう一方の端からとることが多い（下図参照). 二つの表は，ともに大きくなってゆくが，重なることがなければ，表のあふれが生ずることはない.



記号表にハッシュ技法を用いているインタプリタでは，情報が入っていない場所がたくさんあるために大量の記憶場所が無駄に使われている場合が多い．この場所は，そこで解釈実行すべき内部形式となったプログラムの一部をたくわえるのに利用することができる（もちろん，適宜に，余分な分岐命令が入ることになる). これによって解釈実行が多少遅くなるが，問題になるほどではない．また上の場所は，定数や静的な変数（**訳註**：初期値のまま値の変らない変数）にも使用できる.

再入可能コード

プログラム手続きが再入可能であるとは，これが実行中に，他の者が（プログラムの実行を一時中断して)，同じプログラムを実行し始めることができることをいう．これは，マルチプログラミングシステムやタイムシェアリングシステムの環境において最も有用である．このような環境ではいくつかのプログラムが，再入可能なルーチン――例えばSIN――を呼出す可能性がある．どんなに多数のユーザーが一時にそれを実行しようと，主記憶の中には，そのルーチンの写しが一つだけあれば足りる.

コンパイラは，可能ならば，再入可能コードを生成すべきである．コンパイラそのものが再入可能コードでできていることも同様に，いやそれ以上に重要

である．再入可能になっていれば，多くのユーザが同時にコンパイルを始めても，コンパイラの写しはたった一つ記憶装置の上にあればよい．

プログラムが再入可能であるためには，つぎの条件を満たさねばならない：

1. 自分自身を書変えてはならない．すなわち，プログラム中の命令を実行時に変更してはならない．
2. プログラムの実行中に値の変わる変数は全て，そのプログラムの個々の実行に対応するデータ領域の中に場所を割付けなければならない．したがって，このプログラムは，呼出されると，まず，呼出し側のプログラムなり，あるいは自分自身がシステムサブルーチンを呼出すなりして，その実行用のデータ領域の場所を確保する．8.9 節で議論した ALGOL の実行時の管理ルーチンは上の約束に従っていたことに注意．
3. 計算機は，一定個数の高速レジスタを備えているものである．これらのうち1個あるいはそれ以上のレジスタには，各プログラムのデータ領域のいくつかの番地が入っている．主記憶中の変数を参照するときはいつでも，これらのレジスタの一つにある（基底）番地を使用する．プログラムが，割込みで中断されると，システムは，これらのレジスタの内容を退避しなければならない．このプログラムを再開するときは，退避しておいた内容がシステムによってレジスタ上に復原される．

コンパイラのデバッグ

コンパイラをデバッグするための道具は早めに用意しておくべきである．これらのルーチンは，以下のような情報を，文字形式で（8進符号や16進符号ではなく）出力するように作っておくべきである：

1. 構文用スタックと意味用スタック
2. 記号表の各エントリ
3. 内部形式となったプログラム
4. 文字形式と内部形式による最終コード
5. 構文解析で使用された構文規則の跡付け
6. 呼出された意味付けルーチンの跡付け

これらのルーチンの出力が簡潔で読易くなるような設計になるようにかなりの時間をさいてよい．これによってデバッグが一層容易になる．例えば，もし文字形式による最終コードは1命令あたり20文字しか使用しないならば，1ペー

ジに1列最終コードを出力するのではなくて，3～4列にすべきである．

これらのデバッグルーチンは，原始プログラム中の随所に，例えば制御カードのようなものを挿入して，随時，デバッグ用機能を利用したり，利用をやめたりできるようにすべきである．走査部が，このような制御カードを認めて，適宜スイッチを切換えることができる．たった1行の文のコンパイル結果を調べるために全プログラムに対する出力をさせる必要はないので，原始プログラムの部分を限定してデバッグ出力が出せるようにしておくのがよい．

一般に，コンパイラの個々の部分は，論理的な順序：走査部，構文解析部，意味分析部，コード生成の準備部，コード生成部の順にプログラムし検査するものである．実行時の構成はコンパイラ設計にわずかなりとも影響を与えうるので，早目に設計しておくべきである．

コンパイラのテスト・データは作成者以外の人に準備させたい（ソフトウェアの専門業者は新しいプログラムに対しては全てこの方法を採っている）．書き手自身は，無意識のうちにコンパイラの一部しか検査しないようなテストプログラムを作りがちである．

コンパイラが原始プログラムを作表するときは，第1ページの先頭に現在使用中のコンパイラの版の番号，実行の日付など，使用者が，問題に行き当った時に同定が容易であるような情報を印刷すべきである．

このコンパイラはまだデバッグ中であると思っているのなら，そのコンパイラからデバッグ用の機能をはずしてはならない．デバッグ済みということは決してない．

コンパイル時のエラーメッセージ

これらのメッセージは十分明解にして，使用者が特別なエラーメッセージ表を参照してその意味を確かめないですむようにすべきである．簡潔でわかり易いメッセージを与えよ．

よく使われる方法は，各メッセージを句に分割し，各メッセージはその句へのポインタの系列として表現するものである．これによって，記憶場所を節約できる（特定の句は非常によく使われる）ばかりでなく，複雑で手の混んだメッセージを作ることができるようになる．例えば，原始プログラムが解析された後にメッセージを印刷することとし，“メッセージ一覧”の各要素には，

1. エラーメッセージ番号

2. エラーが発生しているカード番号（適用可能な場合）
3. 誤って用いた名前（適用不可ならば0）
4. エラーが発生した時点に入ってくる記号

の情報が入っているとしよう．エラーメッセージ番号は，メッセージを印刷する方法を示した指示書の一覧を参照するのに使用できよう．例えば，メッセージ番号に対する指示書

1. 句1を印刷せよ．
2. カード番号を印刷せよ．
3. 入力文字を印刷せよ．
4. 句2を印刷せよ．
5. 句3を印刷せよ．
6. 名前を印刷せよ．
7. 句4を印刷せよ．

ただし 句1 “カード番号”
句2 “の前に現れている”
句3 “名前”
句4 “は2度宣言されている”

としよう．(1, 20, ABC, ;) に対するメッセージを印刷すれば

> カード番号20 ″;″の前に現れている 名前 ABC は2度宣言されている．

となるだろう．このメッセージを印刷するプログラムは，まさに簡単なインタプリタである．このような方法を用いたコンパイラには PUFFT (Rosen ほか (65) 参照) や ALCOR ILLINOIS 7090 (Gries ほか (65)) などがある．

実行時の誤りの検査とメッセージ

デバッグ用に使わせようというコンパイラは，使用者に役立つような実行時の検査をたくさん入れるべきである．大抵の言語では，

1. 添字付変数の添字の値が範囲を超えたこと
2. ゼロで割るなどの算術演算の検査
3. 値が定義されないうちに変数を参照すること

を検査すべきである．このような検査は，二つの目的に役立つ：一つは，使用者にとって，できるだけ早い時期に実行時の誤りが検出され，もう一つはシステム自身にとって，使用者の誤りによる侵害から逃れることができる．これらの検査はまた適宜選択できるようにしてプログラムのデバッグが済んで実行す

る時点では，検査をしなくてもよいようにすべきである．また，原始プログラムの部分ごとに，デバッグの機能を入れたり抜いたりすることも有用であろう．

このような実行時の検査を実現させることはかなり簡単である．コンパイラがそのような仕事をするコードを生成する．変数が未定義かどうかを決定する検査がおそらく最も難しいものである．計算機の中には各語に“型”を示すビットが付随しているものがいくつかある．“命令”という型の付いた語は算術演算の対象として用いることはできない．したがって，もしコンパイラが全ての語をまず“命令”型に初期設定しておけば，代入が行われる前にその変数を参照しようとすれば，金物の上で割込みが発生する．

このような特徴を持たない計算機では，二つの可能性が存在する：一つは，各変数がテンプレートを持ってそれに値が定義されたかどうかを示すようにするもの（これは記憶場所が余分に要る），もう一つは，各変数に意味のある，しかしまれにしか起らないような値を初期設定し，参照する度にこの値かどうかを検査する．/360 WATFOR では，各変数に初期値X '808080 $\cdots$ 80' を与える（Cress ほか (69) 参照）．もちろん，これは，実際に正しい計算の過程でこの値が生成されたときには誤ったメッセージが印刷されてしまう．

ブロック構造を持った言語では，ブロックや手続きに入るたびに，全ての局所変数は，値を未定義に設定しなければならない．

実行時に印刷されたエラーメッセージには誤りが生じたカードの番号を必ず含めるべきである．これは第16章インタプリタで示したようにして行うことができる．すなわち，NO-OP 命令としてカード番号を挿入しておいてそれにつづいてそのカード上の文に対するコードを生成する．誤りが発生すると，発生した場所の機械語命令から逆にたどって NO-OP 命令を探し出す．

PL/I で用いられている別の方法は原始プログラムのリストの各文に**番号を与える**ものである．各文に対する目的コードは，この文の番号を大域的な変数Eに移すような命令の後につづく．実行時の誤りを検出すると，現に実行していた文の番号がEに入っている．

デバッグ用機能で最善のものは，文字形式のダンプである．これは，誤り発生時点での，参照可能な変数とその値の全ての一覧を**読易い形式**（8進符号や16進符号ではいけない）で出力するものである．第16章ではインタプリタに対するこのようなダンプの手続きを概説した．コンパイラでは，このようなダンプを備えるのは，やや難かしい．何となれば，記号表と，個々のデータ領域の

マップは実行時には手許にないのである．

Bayer ほか (67) は，コンパイラにこのようなダンプを設ける方法を述べている．これをここで手短かに説明しよう．ただしブロック構造を持った言語を仮定しておく．主な点は，コンパイラが，正規の機械語の目的プログラムと同じ長さをもった，L-リストと称せられる一つの目的プログラムを生成するところにある．正規の目的プログラムの各命令に対して，それに対応するL-リストのエントリには，つぎのような情報が含まれている：

1. 原始プログラムデックの対応するカード番号，
2. 命令が生成されているブロックのブロック番号，
3. 現在実行中の文の役割に関する情報——手続き呼出し，添字計算，IF-THEN 判定，など，
4. 原始プログラム演算子，
5. 命令に関わる被演算子の記号表中のエントリ番号（適用可能の場合のみ）．

原始プログラムをコンパイルする時に，コンパイラはL-リストや，記号表やブロック一覧表を2次記憶に出しておく．これらの情報は，実行時に誤りが発生したり，プログラマが特にダンプを要求したりしたときだけに使用される．したがって，ダンプ機能を加えたために実行が遅くなることはない．

実行時に誤りが発生すると，DUMP 手続きが呼出され，それが記憶の割付表（各副プログラムの主記憶上での状態と存在場所の一覧表）を眺めてどの副プログラムで誤りが発生したかを発見する．つぎにその副プログラムに対するL-リストが使用できれば，これと，記号表と，ブロック一覧表を主記憶上に読込んでくる．実行時の誤りを引起した命令に対応するL-リスト上のエントリから，ブロック番号，カード番号などの情報を用いて，第16章に述べたような文字形式のダンプを生成する．

目的プログラム自体に情報を十分残しておくことができるならばL-リストを作成することは，不要かもしれない．しかし，記号表とブロック一覧表は，どうしても必要である．

このような機能が備わっている数少いシステムでは，プログラマは，デバッグの初期の段階で，まさに，ダンプを得るために実行の終りでわざとゼロで割るという知恵を身につけた．

ブートストラップ

原始言語Lに対するコンパイラを機械Mの上で作成するものとしよう．このコンパイラは，L自身で書かれるべきものとする．Lは，コンパイラ作成用言語なのかもしれないし，FORTRAN や ALGOL のような高水準言語かもしれない．このコンパイラを自分自身で書きたい理由は，おそらくアセンブリ言語を除いてほかに記述に適した言語がないからである．この先は，可能な方法が2通りある．

L の小さな部分集合 L_0 に対するコンパイラをアセンブリ言語で書く．この部分集合は非常に小さなもので，コンパイラの実現が容易であるが，つぎの段階で用いるには十分な大きさのものとする．つぎの段階は，L_0 用コンパイラを L_0 で書き換えて検査をする．さて，これをつぎのような段階を踏んでLに至るまで靴のつまみ革（"bootstrap"）を持ち上げるように大きくしてゆく．ステップ i ($i=1, \cdots, n$) において，言語 L_{i-1} に対するコンパイラをこの言語にない特徴を加えてLに近付けた言語 L_i に拡張する．各段階で，その時のコンパイラの古い部分は，この新しい特徴の利点を活かすために，新しい言語で書き直すことができる．L_0 用のコンパイラをいきなり $L=L_n$ 用コンパイラに拡張しないで，小刻みの段階を踏むのは，各段階で，より強力な言語を得て，それで拡張を記述し，それによって作業を容易にするというのが主な理由である．

第二の方法は，別の機械 M′ 上の言語 L′ 用に現存するコンパイラを用いるものである．もし $L'=L$ であれば（言語Lに対するコンパイラはすでに別の機械に作成済みであれば），このコンパイラをLで書き機械M用の コードを生成する．もし $L \neq L'$ であれば，つぎのようにする．L_0 用コンパイラを L′ で記述する（前と同様）．このコンパイラは機械M用のコードを生成する．このコンパイラを機械 M′ 上でコンパイルすると機械M上で動く L_0 用のコンパイラの機械語版が生成される．ここで第一の方法のようにブートストラップを行えばよい．

補　　遺

本書で使用しているプログラム用言語

このプログラム用言語は，ALGOL の拡張であり変種である．ここでこの言語の説明をするのは，本書中のプログラムがどのように実行されるかを説明するだけが目的である．説明は短かいにこしたことはないので，ALGOL の知識を前提とする．できないわけではないが多くの場合厳密な定義はしていない．われわれの目的からいって重要でない細目や，明白なことがらは，読者のプログラム言語に関する知識で埋めてもらいたい．二つの点で解説と例題を多くした．それはポインタの使用と構造を持った変数の使用に関するものである．

名　　前

〈名前〉はつぎのように定義される

〈名前〉::＝〈英字〉{〈英字〉|〈数字〉}

ここで〈数字〉とは文字 0, 1, …, 9 のいずれかであり，〈英字〉とは文字 $, A, B, …, Z, a, b, …, z のいずれかである．〈名前〉は，変数名，手続き名，名札として用いられる．“予約された”名前がいくつかあり，これらの使用法は，普通とは異なる．そのような名前には，ARRAY, BEGIN, BOOLEAN, CASE, CAT, CHARACTER, CONTENTS, DO, ELSE, END, ENDCASE, FALSE, FOR, GOTO, GO TO, IF, INTEGER, POINTER, REAL, RETURN, STEP, STRING, THEN, TRUE, UNTIL, および WHILE がある．

定　　数

〈数字〉0, 1, …, 9 を1個以上並べた系列は，普通どおり，INTEGER 定数である．INTEGER 定数は，16進定数として与えてもよい．例えば，16進定数 X'A' は，定数 10 と同等であり，X'3B' は 59 と同等である．

BOOLEAN 型の定数は TRUE と FALSE である．

もし $C^1, C^2, \cdots, C^n$ が EBCDIC 文字であれば，$'C^1C^2\cdots C^n'$ と $''C^1C^2\cdots C^{n}''$ はともに文字列 $C^1, C^2, \cdots, C^n$ からなる STRING 型定数である．

定数 0 は，POINTER 型定数でもある．これを POINTER 型変数に代入すると，その変数が，指している（参照している）データがないことを意味する．

データの型と変数の宣言

この言語の基本的なデータの型は INTEGER（任意の正負の整数），REAL（任意の実数），BOOLEAN（TRUE または FALSE），CHARACTER（任意の EBCDIC 文字），STRING（空または 1 個以上の文字の系列），および POINTER（他のデータの番地あるいは参照）である．

もし〈型〉が基本的な型の一つであるか，プログラマが定義した構造を持った型（下記）であれば，宣言 D はつぎの形式となる．

D :: =〈型〉〈名前〉,〈名前〉, $\cdots$,〈名前〉

これによってこれらの〈名前〉が，指定した〈型〉の値を代入できる変数の名前であることを示している（例 REAL A, B, C5）．

基本的な型を持ったデータのほかに，プログラムでは，二種の“複合”データを，生成し，参照し，値を変更することができる．宣言

D :: =〈型〉ARRAY〈名前〉{,〈名前〉} [L1 : U1, $\cdots$, L_n : U_n]

によって，これらの〈名前〉が〈型〉型を持った n 次元配列の名前であることを宣言している．ALGOL と同様に，INTEGER 型の式 L_i と U_i によって配列の第 i 次元の下限と上限が与えられる．〈型〉は基本的な型でも，プログラマが定義した構造を持ったデータ型でもよい．上の角かっこ [と] は，超言語記号ではない．

構造を持ったデータ（以下**構造化データ**と称する）は**構成要素**あるいは欄と呼ばれる n 個のデータ ($n>0$) から成っている．構成要素はそれぞれ，名前と〈型〉を持ち，その〈型〉の値を代入することができる．例えば，下図は，B が N, M, Q という名でそれぞれ REAL, INTEGER, POINTER の〈型〉を持った構成要素からなる構造化データであることを示している．

	N (REAL)	M (INTEGER)	Q (POINTER)
B	Nの値	Mの値	Qの値

つぎのような宣言文

STRUCTURE ST (REAL N, INTEGER M, POINTER Q)

を用いれば，ST が新しい構造化データ〈型〉であることを宣言できる．この〈型〉を持ったデータはいずれも上に示すような3個の構成要素から成っている．この型を持つ変数の宣言ができることになる．例えば，B がこの型であること，すなわちBが上図のような形式であることを宣言するには，

ST B

とすればよい．なお構造化データの配列を宣言することができることに注意．

実際には，構造化データや変数を形式的に宣言する必要はないだろう．むしろ，われわれは図表や普通の文章をとおして，どの変数に，構造を持った値を代入し，その構成要素のそれぞれの名前や型が何であるかを説明している．8.6節で，3種の言語における構造化データの使用法を簡単に議論しているので参照してほしい．

名前の有効範囲

ALGOL 型のブロック構造がこの言語でも使用され，変数の割付や，有効範囲を定義している．いくつかの部分プログラムで，ある変数Aが“大域的である”ということが多いが，これは変数Aがそれらの部分プログラムの全てを囲むブロックで宣言されていることを意味している．FORTRAN では，A は COMMON ブロックにある．

手続きの宣言

手続きは ALGOL と同じように宣言される．手続き宣言の構文は

D　　　　　　:: = PROCEDURE 〈手続きの頭部〉;{〈規制部〉;}〈本体〉
〈手続きの頭部〉:: = [〈型〉] 〈名前〉 [(〈引数〉{,〈引数〉})]
〈本　　　　体〉:: = S (任意の文)
〈規　制　部〉:: = 〈型〉 [ARRAY] 〈名前〉 {,〈名前〉}

ここで〈引数〉は名前であり，**仮引数**と呼ばれる．もし〈型〉が〈手続きの頭部〉中に現れれば，この手続きは，関数でこの型の値を関数値とする．手続きの実行中に，関数名を左辺とする代入文を実行し，戻すべき関数値を与えなければならない．規制部によって仮引数の型が記述される．プログラムの前後の関係や普通の説明文の内容から型が明らかな場合は必ずしも書く必要はない．規制部は，仮引数の並びの中に直接記述してしまうこともある（これは，上の構文では示していない）．例えば

```
PROCEDURE X(INTEGER A, B; POINTER P); ···
```

と書けば，**仮引数** A, B が，INTEGER 型で，P が POINTER 型であることを示す．

実引数は，特に指示しなければ全て番地呼び（8.7 節参照）である．手続きの本体の実行を終ると，すなわち RETURN 文が実行されると，制御は，手続きを呼出した点のつぎに戻る．

変数と値：ポインタの用法

1. 名前．もし名前が変数名として宣言されていれば，この名前は〈変数〉を参照している．

　〈変数〉::＝〈名前〉

2. 添字付変数．A は n 次元配列とする．ALGOL と同様に，添字付変数 A $[e^1, e^2, \cdots, e^n]$ は，適当な配列要素を参照する．式 e^i は INTEGER 型でなければならない．

　〈名前〉$(e^1, e^2, \cdots, e^n)$

　という構文も許す．

3. 構造化データの構成要素．B を任意の構造を持った型の〈変数〉としよう．B の構成要素は M, N, ···, Q とする．すると B.M, B.N, ···, B.Q によって，構成要素の〈変数〉をそれぞれ参照できる．したがって構成要素 M の値を（例えば）5 に変更するには，B.M:= 5 を実行すればよい．構成要素 M を参照するのに ″M OF B″ とか ″M(B)″ のような記法を採用している言語もある．8.6 節にこれらの概念を簡単に議論しているので参照されたい．

4. 部分記号列．X を STRING 変数とする．すると，e^1, e^2 を INTEGER

型の式として，

SUBSTR(X, e^1, e^2)

により，X の $e^1, e^1+1, \cdots, e^1+e^2-1$ 番目の文字を値とする STRING 型の変数を表す．なお，X 中の文字は 0 番目から数えることに注意してほしい．したがって，もし X＝'ABCD' ならば，

SUBSTR(X, 1, 2):＝'1X'

を実行して，X を 'A1XD' に変えることができる．

5. 間接番地指定．P を POINTER 型変数，E を任意の式とする．

P:＝@E あるいは P:＝ADDRESS(E)

を実行すると，E を評価した結果の入っている場所の番地が P に代入される．もし E が〈変数〉であれば，P には，その〈変数〉の番地が代入される．そこで，

CONTENTS(P)

を用いて，E を参照することができる．もし E が〈変数〉であれば，

CONTENTS(P):＝3 と E:＝3

は同等であり，これを用いて，変数 E の値を変更することができる．つまり，多くの計算機に備っている間接番地指定の機能がこの言語にあることを示している．

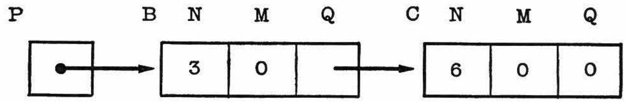
もし P が構造を持った変数 B を参照しており，B には M という名の構成要素があれば，

CONTENTS(P).M の省略形として P.M

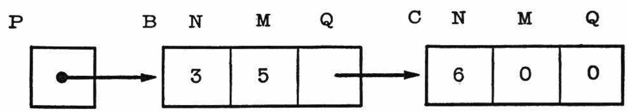
を使用してよいことにする．例を二，三あげよう．P をポインタとし，B と C は，下のように構成要素 N, M, Q からなる構造を持った変数と仮定する：



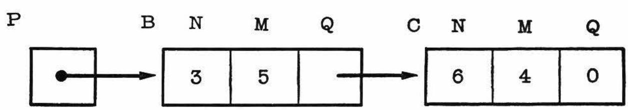
P:＝@B; B.Q:＝@C を実行すると



となる．ここで，矢印は POINTER 型変数 P と B.Q の値を示している．つぎに P.M:＝5 を実行すると



と変る．P.Q.M:＝4 を実行すると



と変る．P.Q は変数 B の構成要素 Q の指している変数を参照している．したがってその値は C の番地である P.Q.M は CONTENTS(CONTENTS(P).Q).M と同等である．

式

式の中に現れる演算はかっこと演算子† の優先順位に注意しながら左から右に評価される．演算子は下記に列挙する（第1行目の演算子の優先順位が最高である）

**
－(単項) @
* / //
＋ －(2項)
＝ ≠ ＜ ≦ ＞ ≧
NOT
AND
OR

† 訳注　JIS ALGOL では，作用素と呼んでいる．

CAT

任意の変数，定数は式である．もしe^1が式であれば(e^1)も式である．

1. 算術式．e^1 および e^2 を算術（INTEGER 型，REAL 型）式とすると，e^1+e^2, e^1-e^2, $-e^1$, e^1*e^2, e^1/e^2, $e^1//e^2$, e^1**e^2 も算術式である．$e^1//e^2$ は，e^1/e^2 を超えない最大の整数値をとる．例えば，$3//2=1$, $4//2=2$, $(-3)//2=-2$ など．
2. 論理式．e^1 と e^2 が論理式であれば，

   e^1 AND e^2, e^1 OR e^2, NOT e^1

   も論理式である．これら3個の演算子は通常の意味を持つ．
3. 関係演算子[†]は，$=$, $\neq$, $<$, $\geq$, $>$, $\leq$ である．e^1, e^2が式であり，Rが関係演算子であれば，関係式[††] e^1 R e^2 は，TRUE または FALSE の値をとる．e^1 が算術式であれば，e^2 も算術式でなければならない．

   もし e^1 が論理式であれば，e^2 も論理式でなければならない．この場合，Rとして意味のあるのは$=$と$\neq$である．もし e^1 が CHARACTER 型または，STRING 型であれば，e^2 も同じものでなければならない．この場合，e^1 と e^2 の短かい方の後ろに空白をつめて同じ長さにした上で，文字の EBCDIC 表現にしたがって比較が行われる．

   もし e^1 が POINTER 型であれば e^2 も同じ型でなければならない．Rは$=$または$\neq$でなければならない．
4. ポインタ式．e^2 を任意の式または変数とすると，@e^1およびADDRESS(e^1)はともに，この式または変数の値の入っている番地を示す．定数0もPOINTER 型の値で，どのようなデータも参照していないことを示している．
5. 構造を持った式．ST をそれぞれ $R_1, R_2, \cdots, R_n$ という型をもった構成要素 $N_1, N_2, \cdots, N_n$ から成る構造を持った型としよう．さらに，$e^1, e^2, \cdots, e^n$をそれぞれ $R_1, R_2, \cdots, R_n$ という型の式とすると，

   $(e^1, e^2, \cdots, e^n)$

は ST 型の式である．e^i はこの式の第 i 要素である．

---

† 訳注　JIS ALGOL では，比較作用素と呼んでいる．
†† 訳注　JIS ALGOL では，比較式と呼んでいる．

文[†]

文は，分岐によって実行順序が変更されないかぎり，並んでいる順序に実行される．文の種類は以下のとおり．ここで "S" は "文" を意味する非終端記号である．

1. 名札付き文．S ::= 〈名札〉: S

任意の文に〈名札〉を付けることができる．〈名札〉は〈名前〉か符号のない整数でなければならない．

2. 空文．　　S ::=

これによってプログラムを書く際に融通性が増し，構文の記述がすっきりする．

3. 代入文．　　S ::= 〈変数〉:= 〈式〉

これは ALGOL と同様に実行される．型の変換は自動的に行われる．〈変数〉の番地が計算され，〈式〉の値が評価されて，その結果が，変数に代入される．

4. 分岐[††]．　　S ::= GOTO 〈名札〉|GO TO 〈名札〉

実行順序を〈名札〉の付いている文に変更するもの．

5. 条件文．　　S ::= IF 〈式〉 THEN S^1 [ELSE S^2]

ここで〈式〉は評価の結果，TRUE または FALSE となる論理式である．もし TRUE であれば，S^1 が実行される．もし FALSE であり，S^2 が存在すれば S^2 が実行される．ELSE は直前の THEN と対応する．

6. 複合文．　　S ::= BEGIN S {; S} END

文Sが順次実行される．複合文は，いくつかの文をひとまとめにするために使用される．

7. ブロック．　　S ::= BEGIN D; {D;} S {; S} END

これは ALGOL のブロックである．宣言Dが少くとも1個なければならない．ブロックに入ると，これらの宣言中で宣言されている変数に記憶場所を割当て

---

† 訳注　文（statement）は2章〜7章で用いている文（sentence）と区別するために本文中では実行文ということばをあてていることがある．

†† 訳注　JIS ALGOL では飛越し文と呼ぶ．

る．そして，文が順次実行される．ブロックからの脱出にあたってこのブロックの中で変数に割当てられた記憶場所は，解放される．ブロックからの脱出は，分岐，RETURN 文あるいは，ブロック中の最後の文の実行が終った時に生じる．

8. WHILE 文．S :: = WHILE 〈式〉 DO S¹

これはつぎの文と等価である：

AGAIN: IF 〈式〉 THEN BEGIN S¹; GO TO AGAIN END

6. 繰返し文．S :: = FOR 〈変数〉 : = 〈式〉 STEP 〈式〉
UNTIL 〈式〉 DO S¹

これは ALGOL と同じ．

10. CASE 文．S :: = CASE 〈式〉 OF S {;S} ENDCASE
| CASE 〈式〉 OF BEGIN S {;S} END

このように，CASE 文には等価な 2 通りの形式がある．CASE 文はつぎのように実行される：〈式〉が評価されて，整数値 i が得られる．文の並びの中から i 番目の文が実行される．つぎに実行の制御は，CASE 文の直ぐつぎに移る．もし CASE 文中の文の並びのいずれかに対して分岐があると，その文を実行して，CASE 文の直後の点に制御が移る．

11. 手続き呼出し，関数呼出し．

S :: = 〈名前〉 [(〈引数〉, 〈引数〉, ･･･, 〈引数〉)]

引数は，任意の式または配列名でよい．これらは指定されないかぎり，番地呼び（8.7 節参照）である．呼出しに際して実引数と仮引数の対応は FORTRAN や ALGOL のように行われる．手続きは，再帰的であってよい．標準関数が二つあり，プログラム中で用いられる．それらの定義はつぎのとおり：

SIGN(X) = IF X < 0 THEN −1 ELSE IF X = 0 THEN 0
ELSE +1

MAXIMUM(X, Y) = IF X < Y THEN Y ELSE X

12. RETURN 文．　　S :: = RETURN

呼出した点へ戻る．手続きの本体の中だけに現れる．

参　考　文　献

省略記号はつぎのとおりである：

CACM	Communications of the ACM
JACM	Journal of the ACM
NACM	Proceedings of the National ACM Conference
SJCC	Proceedings of the Spring Joint Computer Conference
FJCC	Proceedings of the Fall Joint Computer Conference
EJCC	Proceedings of the Eastern Joint Computer Conference
WJCC	Proceedings of the Western Joint Computer Conference
AnnR	*Annual Review in Automatic Programming,* Pergammon Press
Comp J	Computer Journal
IFIP	Proceedings of the IFIP Congress

同一著者の文献が2編以上になる場合は，著者名(出版年)の形で示す．さらに同じ年度に2編以上の文献が出版されている場合は，年代の後に，a, b, …を付けて区別する．

Allard, R. W., Wolf, K. A., and Zemlin, R. A. Some effects of the 6600 Computer on language structures. CACM 7 (Feb. 1964), 112-127.

Allen, F. E. Program optimization. In AnnR 5 (1969), 239-307.

Anderson, J. P. A note on compiling algorithms. CACM 7 (March 1964), 149-150.

Arden, B. W., Galler, B. A. and Graham, R. M. (61) An algorithm for equivalence declarations. CACM 4 (July 1961), 310-314.

___, ___, and ___. (62) An algorithm for translating Boolean expressions. JACM 9 (April 1962), 222-239.

Backus, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. International Conf. on Information Processing, UNESCO (1959), 125-132.

___, et al. The FORTRAN automatic coding system. WJCC (1957), 188-198.

Baer, J. L., and Bovet, D. P. Compilation of arithmetic expressions for parallel computations. IFIP 68, B4-B10.

Batson, A. The organization of symbol tables. CACM 8 (Feb 1965), 111-112.

Bauer, H., Becker, S., and Graham, S. ALGOL W implementation. CS 98, Computer Science Dept., Stanford Univ., 1968.

Bayer, R., Gries, D., Paul, M., and Wiehle, H. R. The ALCOR ILLINOIS 7090/7094 post mortem dump. CACM 10 (Dec 1967), 804-808.

Belady, L. A. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5 (1966), 78-82.

Bell, J. R. (69) A new method for determining linear precedence functions for precedence grammars. CACM 12 (Oct 1969), 567-569.

___. (70) Quadratic quotient method: a hash code eliminating secondary clustering. CACM 13 (Feb 1970), 107-109.

Berman, R., Sharp, J., and Sturges, L. Syntactical charts of COBOL 61. CACM 5 (May 1962), 260.

Berns, G. M. Description of FORMAT, a text-processing program. CACM 12 (March 1969), 141-146.

Breuer, M. A. Generation of optimal code for expressions via factorization. CACM 12 (June 1969), 333-340.

Brooker, R. and Morris, D. A general translation program for phrase structure languages. JACM 9 (Jan 1962), 1-10.

Brooker, R. et al. The compiler-compiler. AnnR 3 (1963), 229-275.

___, Morris, D., and Rohl, J. S. (62) Trees and routines. Comp J 5 (April 1962), 33.

___, ___, and ___. (67) Experience with the compiler compiler. Comp J 9 (1967), 345-349.

Brown, P.J. (67) The ML/1 macro processor. CACM 10 (Oct 1967), 618-623.

___. (69) A survey of macro processors. AnnR 6 (1969), 37-88.

Busam, V. and Englund, D. Optimization of expressions in FORTRAN. CACM 12 (Dec 1969), 666-674.

Cheatham, T. E. The TGS-II translator-generator system. IFIP 1965, 592-593.

___, and Sattley, K. Syntax directed compiling. SJCC 1964, 31-57.

Chomsky, N. (56) Three models for the description of language. IREE Trans. Inform. Theory, vol. IT2, (1956), 113-124.

___. (59) On certain formal properties of grammars. Information and Control 2, (1959), 137-167.

___. (63) Formal Properties of Grammars. In Handbook of Mathematical Psychology, Vol. 2, Luce, Bush and Galanter (Eds.) John Wiley and Sons, Inc., New York, 1963, 323-418.

Cocke, J. and Schwartz, J. T. Programming languages and their compilers. Preliminary notes. Courant Inst. of Math. sciences. New York Univ., 1970.

Cohen, D. J., and Gotlieb, C. C. A list structure form of grammars for syntactic analysis. Computing Surveys 2 (March 1970), 65-81.

Colmerauer, A. Total precedence relations. JACM 17 (Jan 1970), 14-30.

Conway, M. E. Design of a separable transition diagram compiler. CACM 6 (July 1963), 396-408.

Conway, R. W. and Maxwell, W. L. CCRC - the Cornell computing language. CACM 6 (June 1963), 317-321.

___, et al. PL/C. A high performance subset of PL/I. TR 70-55, Computer Science Dept. Cornell Univ., 1970.

Cress, P. H. et al. Description of /360 WATFOR. Dept. of Applied Analysis and Computer Science, Univ. of Waterloo. Report No. CSTR-1000.

Dantzig, G. B. and Reynolds, G. Optimal assignment of computer storage by chain decomposition of partially ordered sets. Univ. of Calif., Berkeley. Operations research center Rpt. No. ORC-66-6, March 1966.

DeRemer, F. Simple LR(k) grammars. Computer Science Dept., Univ. of Calif at Santa Cruz, May 1970.

Dijkstra, E. W. An ALGOL 60 translator for the X1. AnnR 3 (1963), 329-356.

Domolki, B. An algorithm for syntactic analysis. Computational Linguist. 3 (1964), 29-46. (Hungarian Journal).

Earley, J. Generating a recognizer for a BNF grammar. Technical Report, Carnegie Inst. of Tech., Pittsburgh, Penn. June 1965.

Ehrman, J. Q-type address constants, dummy external symbols, and pseudo registers. CGTM 66, Stanford Linear Accelerator Center. April 1969.

Eickel, J. Generation of parsing algorithms for Chomsky 2-type languages. Report No. 6401. Math. Inst. der Tech. Hochschule, Munich, 1964.

___, Paul, M., Bauer, F. L., and Samelson, K. A syntax controlled generator of formal language processors. CACM 6 (Aug 1963), 451-455.

Evans, A. An Algol 60 Compiler. AnnR 4 (1964), 87-124.

Fateman, R. J. Optimal code for serial and parallel computation. CACM 12 (Dec 1969), 694-695.

Feldman, J. A. A formal semantics for computer languages and its application in a compiler-compiler. CACM 9 (Jan 1966), 3-9.

___, and Gries, D. Translator writing systems. CACM 11 (Feb 1968), 77-113.

Fischer, M. J. Some properties of precedence languages. Proceedings ACM sympcsium on theory of computing (May 1969), 181-188.

Flores, I. Computer Sorting. Prentice-Hall, Inc. Englewood Cliffs. 1969.

Floyd, R. W. (61a) An algorithm for coding efficient arithmetic operations. CACM 4 (Jan 1961), 42-51.

___. (61b) A descriptive language for symbol manipulation. JACM 8 (Oct 1961), 579-584.

___. (63) Syntactic analysis and operator precedence. JACM 10 (July 1963), 316-333.

___. (64a) Bounded context syntactic analysis. CACM 7 (Feb 1964), 62-67.

___. (64b) The syntax of programming languages -- a survey. IEEE Trans. EC13, 4 (Aug 1964), 346-353.

Freeman, D. Error correction in CORC: the Cornell computing language. Thesis, Cornell Univ., 1963.

Galler, B., and Perlis, A. J. A proposal for definitions in ALGOL. CACM 10 (April 1967), 204-219.

Garwick, J. V. GARGOYLE, a language for compiler writing. CACM 7 (Jan 1964), 16-20.

Gear, C. W. (64) Optimization of the address field computation in the ILLIAC II assembler. Comp J 6 (Jan 1964), 332.

___. (65) High speed compilation of efficient object code. CACM 8 (Aug 1965), 483-488.

Gilbert, P. On the syntax of algorithmic languages. JACM 13 (Jan 1966), 90-107.

Gill, A. Introduction to the Theory of Finite-State Machine. McGraw-Hill Book Co., New York, 1962.

Ginsburg, S. (62) An Introduction to Mathematical Machine Theory. Addison-Wesley, Reading, Mass., (1962).

___. (66) The Mathematical Theory of Context-Free Languages. McGraw-Hill Bock Company, N. Y., 1966.

Glennie, A. E. On the syntax machine and the construction of a universal compiler. Techn. Rpt. No. 2, Computation Center, Carnegie Inst. of Tech. (1960).

Gorn, S. Specification languages for mechanical languages and their processors, a baker's dozen. CACM 4 (Dec 1961), 532-542.

Gower, J. C. The handling of multiway tables on computers. Comp J 4, (Jan 1962), 280-286.

Grau, A. A. Recursive processes and ALGOL translation. CACM 4 (Jan 1961), 10-15.

Greenwald, I. A technique for handling macro instructions. CACM 2 (Nov 1959), 21-22.

___, and Kane, M. The SHARE 709 system: programming and modification. JACM 6 (1959), 396.

Gries, D. The use of transition matrices in compiling. CACM 11 (Jan 1968), 26-34.

___, Paul, M., and Wiehle, H. R. Some techniques used in the ALCOR ILLINOIS 7090. CACM 8 (Aug 1965), 496-500.

Griffiths, T. V., and Petrick, S. R. On the relative efficiencies of context-free grammar recognizers. CACM 8 (May 1965), 289-299.

Halstead, M. H. Machine-Independent Computer Programming. Spartan Books, Washington D. C. 1962.

Halpern, M. (64) XPOP: a metalanguage without metaphysics. FJCC 1964, 57-68.

___. (67) A manual of the XPOP programming system. Electronic Sciences Lab., Lockheed Missiles and Space Company, Palo Alto, California, March 1967.

___. (68) Toward a general processor for programming languages. CACM 11 (Jan 1968), 15-25.

Hellerman, H. (62) Addressing multidimensional arrays. CACM 5 (April 1962), 205-207.

___. (66) Parallel processing of algebraic expressions. IEEE Trans. EC-15 (Feb 1966), 82-91.

Hext, J., and Roberts, P. Syntax analysis by Domolki's algorithm. Comp J 13 (Aug 1970), 263-271.

Hill, V., Langmaack, H., Schwarz, H. R., and Seegmuller, G. Efficient handling of subscripted variables in ALGOL 60 compilers. Proc. Symbolic Languages in Data Processing, Gordon and Breach, New York, 1962, 331-340.

Hoare, C. A. R. Record handling. In Programming Languages. Academic Press, 1968, pages 291-347.

Hopcroft, J. and Ullman, J. Formal languages and their Relation to Automata. Addison-Wesley, New York, 1969.

Hopgood, F. R. A. Compiling Techniques. American Elsevier, Inc., New York 1969.

Horning, J. J. Empirical comparison of LR(k) and precedence parsers. Memorandum. Computer research Group, Univ. of Toronto. Aug 1970.

Horwitz, L. P. et al. Index register allocation. JACM 13 (Jan 1966), 43-61.

Huskey, H. D. Compiling techniques for algebraic expressions. Comp J 4 (1961), 10-19.

___, Love, R., and Wirth, N. A syntactic description of BC NELIAC. CACM 6 (July 1963), 367-375.

Huxtable, D. H. R. On writing an optimizing translator for ALGOL 60. In Introduction to System Programming. Academic Press Inc., New York, 1964.

IBM. (a) FORTRAN (H) Compiler, Programming Logic Manual. Form Y28-6642.

IBM. (b) PL/I (F) Compiler, Programming Logic Manual. Form Y28-6800.

IBM. (c) An Introduction to the Compile-Time Facilities of PL/I. Technical Publication C20-1689. IBM Corporation. August, 1968.

IBM. (d) Operating System /360 Assembler Language. IBM Form C28-6514.

Ichbiah, J. D. and Morse, S. P. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. CACM 13 (Aug 1970), 501-508.

Ingerman, P. Z. (61a) Thunks. CACM 4 (Jan 1961), 55-58.

___. (61b) Dynamic declarations. CACM 4 (Jan 1961), 59-60.

___. (66) A Syntax Oriented Translator. Academic Press, New York, 1966.

Irons, E. T. (61a) A syntax directed compiler for ALGOL 60. CACM 4 (Jan 1961), 51-55.

___. (63a) The structure and use of the syntax-directed compiler. AnnR 3 (1963), 207-227.

___. (63b) An error correcting parse algorithm. CACM 6 (Nov 1963), 669-673.

___. (64) "Structural connections" in formal languages. CACM 7 (Feb 1964), 67-72.

___, and Feurzeig, W. Comments on the implementation of recursive procedures and blocks in ALGOL 60. CACM 4 (Jan 1961), 65-69.

Johnson, L. R. Indirect chaining method for addressing on secondary keys. CACM 4 (May 1961), 218-222.

Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. Automatic generation of efficient lexical processors using finite state techniques. CACM 11 (Dec 1968), 805-813.

Kanner, H. An algebraic translator. CACM 2 (Oct 1959), 19-22.

Kent, W. Assembler-language programming. Computing Surveys 1 (Dec 1969), 183-196.

Kleene, S. C. Representation of events in nerve-sets, in Automata Studies pp. 3-42. Princeton Univ. Press, Princeton, N.J. 1956.

Knuth, D. E. (59) Runcible - algebraic translation on a limited computer. CACM 2 (Jan 1959), 18-21.

___. (65) On the translation of languages from left to right. Inf. Contr. 8 (Oct 1965), 607-639.

___. (68) The Art of Computer Programming. Vol 1. Addison-Wesley Publishing Company, Inc. Reading, Mass., 1968.

___. (70) The Art of Computer Programming. Vol. 3. Addison-Wesley Publishing Company, Inc. Reading, Mass., 1970.

Kuno, S. and Oettinger, A. G. Multiple-path syntactic analyzer. Information Processing 62 (IFIP Congress), Popplewell (Ed.), North Holland Publishing Co., Amsterdam, 1962, 306-311.

Leavenwirth, B. M. (64) FORTRAN IV as a syntax language. CACM 7 (Feb 1964), 72-80.

___. (66) Syntax macros and extended translation. CACM (Nov 1966), 790-793.

Lewis, P. M. II and Stearns, R. E. Syntax-directed transductions. JACM 15 (July 1968), 465-488.

Loeckx, J. An algorithm for the construction of bounded-context parsers. Report R99. Manufacture Belge de Lampes et de Materiel Electronique. March 1969.

Lowry, E. and Medlock, C. Object code optimization. CACM 12 (Jan 1969), 13-22.

Luccio, F. A comment on index register allocation. CACM 10 (Sept 1967), 572-574.

Lynch, W. C. A high-speed parsing algorithm for ICOR grammars. Report No. 1097, Andrew Jennings Computing Center, Case Western Reserve Univ., 1968.

Martin, D. F. Boolean matrix methods for the detection of simple precedence grammars. CACM 11 (Oct 1968), 685-687.

Maurer, W. D. An improved hash code for scatter storage. CACM 11 (Jan 1968), 35-38.

McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. CACM 4 (April 1960), 184-195.

McClure, R. M. TMG -- a syntax-directed compiler. NACM 20 (1965), 262-274.

McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophysics 5, 115-333.

McIlroy, M. D. (60) Macro instruction extensions of compiler languages. CACM 3 (April 1960), 214-220.

___. (63) A variant method of file searching. CACM 6 (March 63), 101.

McKeeman, W. M. (65) Peephole optimization. CACM 8 (July 1965), 443-444.

___. (66) An approach to language design. CS 48, Computer Science Dept., Stanford Univ., 1966.

___, Horning, J. J., and Wortman, D. B. A Compiler Generator implemented for the IBM System/360. Prentice Hall, 1970.

Mendicino, S. and Zwackenberg, R. A FORTRAN code optimizer for the CDC 6600. UCRL-14162, Lawrence Radiation Lab, April 1965.

Mooers, C., and Deutsch, L. P. TRAC, a text handling language. NACM 20 (1965), 229-246.

Moore, E. F. Gedanken experiments on sequential machines. Automata Studies, pp 129-153. Princeton Univ. Press, Princeton, 1956.

Morgan, H. L. Spelling correction in system programs. CACM 13 (Feb 1970), 90-94.

Morris, R. Scatter storage techniques. CACM 11 (Jan 1968), 38-44.

Nakata, I. A note on compiling algorithms for arithmetic expressions. CACM 10 (Aug 1967), 492-494.

Naur, P. (63a) (Ed.). Revised report on the algorithmic language ALGOL 60. CACM 6, (Jan 1963), 1-17.

___. (63b) The design of the GIER ALGOL compiler. BIT 3 (1963), 124-140, 145-166.

___. (65) The design of the GIER ALGOL compiler. AnnR 4 (1965), 49-85.

Parikh, R. J. Language generating devices. M.I.T. Research Lab. Electron. Quart. Prog. Report. 60, 1961, 199-212. (Also in JACM 13 (Oct 1966), 570-581, under the title: On context free languages.)

Paul, M. ALGOL 60 processors and a processor generator. Proc. IFIP Congress, Munich, 1962, 493-497.

Peterson, W. W. Addressing for random-access storage. IBM Journal of Research and Development 1 (1957), 130.

Plaskow, J. and Schumann, S. The TRANGEN system on the M460 computer. AFCRL-66-516 (July 1966).

Post, E. L. A variant of a recursively unsolvable problem. Bull. Am. Math. Soc., 52 (1946), 264-268.

Purdon, P. W. A transitive closure algorithm. Computer Sciences Technical Report # 33, July 1968. Univ. of Wisconsin.

Radke, C. E. The use of quadratic residue search. CACM 13 (Feb 1970), 103-105.

Randell, B. and Russell, D .J. ALGOL 60 Implementation. Academic Press, London, 1964.

Redziejowski, R. R. On arithmetic expressions and trees. CACM 12 (Feb 1969), 81-84.

Reynolds, J. C. An introduction to the COGENT Programming System. NACM 20 (1965), 422-436.

Rishel, W. J. Incremental compilers. Datamation. Also in Software for Computer Systems, College Readings, Inc. Arlington, Va. 1970.

Rosen, S. A compiler-building system developed by Brooker and Morris. CACM 7 (July 1964), 403-414.

___, Spurgeon, R. A., and Donnelly, J. K. PUFFT - the Purdue University fast FORTRAN translator. CACM 8 (Nov 1965), 661-666.

Rosenkrantz, D. S. and Stearns, R. E. Properties of deterministic top down grammars. ACM Symposium on Theory of Computing (May 1969), 165-180.

Ross, D. (66) AED bibliography. Memo MAC-M-278-2, Project MAC, M.I.T., Cambridge, Mass., Sept. 1966.

___. (67a) The AED approach to generalized computer-aided design. NACM 1967, 367-385.

___. (67b) The AED free storage package. CACM 10 (Aug 1967), 481-492.

Rutishauser, H. Automatische Rechenplanfertigung bei programm-gesteuerten Rechenmaschinen. Mitt. Inst. fur Angew. Math. der ETH Zurich, Nr. 3, 1952.

Samelson, K., and Bauer, F. L. Sequential formula translation. CACM 3 (Feb 1960), 76-83.

Sattley, K. Allocation of storage for arrays in ALGOL 60. CACM 4 (Jan 1961), 60-65.

Schay, G. and Spruth, W. G. Analysis of a file addressing method. CACM 5 (Aug 1962), 459-462.

Schneider, F. W. and Johnson, G. D. Meta-3; A syntax-directed compiler writing compiler to generate efficient code. NACM 19 (1964).

Schorre, D. V. META II: A syntax-oriented compiler writing language. NACM 19 (1964), page D1.3.

Shaw, A. Lecture notes on a course in systems programming. CS 52, Computer Science Dept., Stanford Univ., 1966.

Shaw, C. J. A specification of Jovial. CACM 6 (Dec 1963), 721-735.

Sethi, R. and Ullman, J. D. The generation of optimal code for arithmetic expressicns. JACM 17 (Oct 1970), 715-728.

Sheridan, P. B. The FORTRAN arithmetic-compiler of the IBM FORTRAN automatic coding system. CACM 2 (Feb 1959), 9.

Sibley, R. A. The SLANG system. CACM 4 (Jan 1961), 75-84.

Squire, J. S. A translation algorithm for a multiple processor computer. NACM 1963.

Standish, T. A. A data definition facility for programming languages. Computer Science Rpt., Carnegie Inst. of Tech., May 1967.

Stearns, R. E. and Lewis II, P. M. Property grammars and table machines. IEEE Conference record of the 1968 annual symp. on switching and automata theory, 106-119.

Stone, H. S. One-pass ccmpilation cf arithmetic expressions for a parallel processor. CACM 10 (April 1967), 220-223.

Strachey, C. A general purpose macrogenerator. Comp J 8 (1965-1966), 225-241.

Taylor, W., Turner, L., and Waychoff, R. A syntactical chart of ALGOL 60. CACM 4 (Sept 1961), 393.

Tixier, V. Recursive functions of regular expressions in language analysis. CS 58. Computer Science Dept., Stanford Univ., March 1967.

Unger, S. A global parser for context-free phrase structure grammars. CACM 11 (April 1968), 240-247.

Van der Poel, L. W. The construction of an ALGCL translator for a small computer. In Prog. Lang. in Data Processing, 229-236. Gordon and Breach, 1963.

Wagner, R. A. Scme techniques for algebraic optimization with application to matrix arithmetic expressions. Thesis. Carneqie-Mellon Univ., June 1968.

Waite, W. A language-independent macro processor. CACM 10 (July 1967), 433-440.

Warshall, S. (61) A syntax directed generator. EJCC 1961, 295-305.

___. (62) A theorem on Boolean matrices. JACM 9 (Jan 1962), 11-12.

___, and Shapiro, R. M. A general-purpose table-driven compiler. SJCC 1964, 59-65.

Wegner, P. (68a) Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, New York, 1968.

Whitney, G. An extended BNF for specifying the syntax of declarations. SJCC 1969, 801-812.

Wirth, N. and Weber, H. Euler -- a generalization of ALGOL and its definition. Part I, Part II. CACM 9 (Jan, Feb 1966), 13-25, 89-99.

___, and Hoare, C. A. R. A contribution to the development of ALGOL. CACM 9 (June 1966), 413-432.

Yershov, A. L. (58) On programming of arithmetic operations. CACM 1 (Aug 1958), 3-6.

___. (66) ALPHA - an automatic programming system of high efficiency. JACM 13 (Jan 1966), 17-24.

Zimmer, R. Weak precedence. Proc. of the International Computing Symposium 1970 / ACM Europe, Part II (May 1970), 576-587.

索　　引

以下の索引中で，見出し“変数名”のもとに，本書中で使用している変数名(英字名)が列挙してある．ページ番号は，その名が定義または使用されている場所を示す．同様に，“手続き名”のもとに，本書中で使用している関数やサブルーチンの名，“コンパイラ名”のもとに，本書中で引用している現実のコンパイラの名を，それぞれ一覧できるようにまとめてある．

英字索引および人名索引は事項索引の末尾に記載されている．

事 項 索 引

あ 行

い

う

え

に

ね

の

は 行

ひ

ふ

英字索引

人 名 索 引

M E M O

M E M O

MEMO

MEMO

《訳者略歴》

牛島和夫
うしじま かずお

1937年　東京都に生まれる
1961年　東京大学工学部応用物理学科(数理コース)卒業
1963年　同 大学院数物系研究科修士課程修了
1963年　九州大学中央計数施設勤務
1971年　九州大学工学部情報工学科助教授，工学博士
1977年　同 教授
現　在　九州大学工学部情報工学科教授・工学博士

コンパイラ作成の技法

昭和53年7月10日　初版発行

訳　者　牛　島　和　夫
発行者　村　越　勝　弘
発行所　**日本コンピュータ協会**
東京都文京区本郷2丁目15—13
発売元　**科学技術出版社**
東京都文京区本郷2丁目15—13
〒113　電話(03) 813—6920番

検印廃止